POO: Os mandamentos da Programação Orientada a Objetos e Design de Software

Clube Delphi +PHP

DevMedia group

Ano 9 :: Edição 117 :: R$ 11,90

Boas Práticas
Crie um agendador de tarefas usando técnicas de POO e Padrões de Projeto - Parte 2

Introdução ao DataSnap 2010

Primeiros passos no desenvolvimento multicamadas

Brinde

Comprando esta edição ganhe 4 vídeo-aulas:

1 – Explore a nova RTTI para desenvolver um framework de mapeamento O/R – Parte 1
2 – Explore a nova RTTI para desenvolver um framework de mapeamento O/R – Parte 2
3 – Trabalho em equipe com StarTeam – Gerenciando Tarefas – Parte 1
4 – Trabalho em equipe com StarTeam – Gerenciando Tarefas – Parte 2

Quick Update
Autenticação no novo DataSnap

PHP
Mapeamento Objeto / Relacional

ISSN 1517990-7

Cursos Online

Web Mobile PLUS

A **Revista WebMobile** oferece para seus assinantes uma série de Cursos Online de alto padrão de qualidade **.Conheça abaixo os cursos já disponíveis.**

Curso de .net em destaque

Aplicação completa de orçamento Doméstico no Visual Studio 2005

Confira neste curso online como criar uma aplicação completa no Visual Studio 2005, usando ASP.NET, Web Services, Mobile e muito mais! Veja como criar uma aplicação de orçamento doméstico, usando diagrama de classes de uma forma muita produtiva, criar classes de negócios de acesso a dados.

Confira o plano de aula completo:
www.devmedia.com.br/domesticovs

Curso de Java em destaque

Introdução ao desenvolvimento para celulares com J2ME

Confira neste curso os principais recursos do J2ME. Aprenda também o passo a passo para criar sua primeira aplicação J2ME. Neste curso você irá aprender diversas funcionalidades desta tecnologia para desenvolvimento de dispositivos móveis.

Confira o plano de aula completo:
www.devmedia.com.br/celularesj2me

Assine a **WebMobile** e comece já seu treinamento!
www.devmedia.com.br/assine

A sua melhor opção de aprendizagem!

Em breve mais novidades para você! www.devmedia.com.br/curso

DevMedia group Mais Informações: www.devmedia.com.br/central - Tel.: 21 3382-5038

Sumário

Quick Update

7 – Autenticação no DataSnap 2010
Implemente segurança em aplicações multicamadas de uma forma simples
[*Rodrigo Carreiro Mourão*]

Boas Práticas

Boas Práticas – Engenharia de Software

10 – Mandamentos da Orientação a Objetos
Conheça os bons princípios que podem garantir qualidade
[*Paulo Quicoli*]

Boas Práticas

20 – Crie um agendador de tarefas 00 – Parte 2
Refatorando o código
[*Rafael Stavarengo*]

Boas Práticas - Expert

29 – Design Patterns para o mundo real
Template Method, Abstract Factory e Adapter
[*Ricardo Coelho*]

Easy Delphi

Easy Delphi – Tutorial

37 – Primeiros passos com o novo DataSnap 2010
Conheça as novidades na prática
[*Fabrício Hissao Kawata*]

PHP

PHP

45 – Mapeamento Objeto / Relacional
Aumente a elegância no desenvolvimento sem perder produtividade
[*Rodrigo Carreiro Mourão*]

Brinde na web desta edição - 4 vídeos! Confira:

4 Vídeos

- **Mapeamento objeto / relacional no Delphi 2010 - Parte 1 –** Explore a nova RTTI para desenvolver um framework de mapeamento objeto / relacional
http://www.devmedia.com.br/articles/viewcomp.asp?comp=15984

- **Mapeamento objeto / relacional no Delphi 2010 - Parte 2 –** Nesta segunda parte veja como criar as classes responsáveis pela conexão com o banco de dados
http://www.devmedia.com.br/articles/viewcomp.asp?comp=15985

- **StarTeam – Distribuindo tarefas aos membros da equipe - Parte 1 –** Conheça e utilize o item Tasks do StarTeam
http://www.devmedia.com.br/articles/viewcomp.asp?comp=15833

- **StarTeam – Distribuindo tarefas aos membros da equipe - Parte 2 –** Veja como monitorar o progresso da realização da tarefa, fazer vínculos entre check-ins de arquivos e tarefas e também como criar uma Change Request para reportar um problema na realização da tarefa
http://www.devmedia.com.br/articles/viewcomp.asp?comp=15901

Para acessar os vídeos utilize os seguintes dados
Login: **DVM.CD** *Senha:* **dteam**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Para votar, você vai precisar do código e senha de banca desta edição, que é:
Login: **DVM.CD** Senha: **dteam**

Clube Delphi +PHP

Ano 9 - 117ª Edição 2010 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Paulo Quicoli
pauloquicoli@devmedia.com.br

Daniel Laporte
daniel.laporte@gmail.com

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br

Equipe Editorial
Vitor Luiz Rubio
vitorrubio@gmail.com

Rafael Stavarengo
faelsta@gmail.com

Capa e Diagramação
Romulo Araujo
romulo@devmedia.com.br

Coordenação Geral
Daniella Costa - daniella@devmedia.com.br

Revisão e Supervisão
Thiago Vincenzo - thiago.v.ciancio@devmedia.com.br

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Cristiany Queiroz – Atendimento ao Leitor
www.devmedia.com.br/mancad
(21) 3382-5038

Kaline Dolabella – Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5038

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site e para fechar parcerias ou ações específicas de marketing com a DevMedia, entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor!

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

Esta edição destaca os "mandamentos da orientação a objetos", os princípios e boas práticas que todo desenvolvedor deve saber. No artigo de Paulo Quicoli, você conhecerá o que são os princípios SOLID, um conjunto de regras que devem ser seguidas para se obter um software de qualidade: princípio da responsabilidade única, o design de classes abertas para extensão e fechadas para modificação, inversão de dependência, programação para interfaces e dependência somente de classes abstratas, são dicas preciosas que você deve ter em mãos quando modelar seu sistema OO em Delphi. E como sempre, você verá tudo isso aplicado na prática, em situações reais.

Nosso outro destaque vai para o DataSnap. O Delphi 3 introduziu o suporte ao desenvolvimento multicamadas, na tecnologia chamada na época de MIDAS (Multitier Distributed Application Services). O Delphi 4 aprimorou este suporte, sendo considerado por muitos como sendo a primeira versão "oficial" a permitir o desenvolvimento multitier. No Delphi 5 tivemos um grande avanço, a inclusão da interface IAppServer, que facilitou a comunicação entre cliente e servidor de aplicação, principalmente para ambientes state-less, como COM+. O Delphi 6 rebatizou o MIDAS para DataSnap, para fazer trio com as tecnologias WebSnap e BizSnap. O DataSnap depois disso deu algumas voltas pelo universo .NET, através da tecnologia .NET Remoting. Chegando agora em 2010, através do Delphi 2010, a tecnologia está totalmente reestruturada, desamarrada de protocolos proprietários e suportando padrões consolidados e abertos do mercado, como HTTP, TCP/IP, REST e JSON. Um fator que sempre me chamou a atenção em todos esses anos de desenvolvimento multicamadas com o Delphi foi a completa transparência com relação a protocolos. Passamos pelo COM, DCOM, MTS, COM+, Sockets, SOAP, Corba, Remoting, sem nunca mesmo precisar conhecer os princípios e implementações de cada um, apesar de existirem classes concretas para usar cada protocolo. Com o DataSnap 2010 (e 2009), mais um passo foi dado além, o que pessoalmente considero ser uma das melhores novidades: passamos a usar dbExpress, através de sua arquitetura de drivers, no lado cliente da aplicação. Além dessa abstração, ganhamos produtividade e diminuímos a curva de aprendizado e tempo para portar aplicações 2-tier para 3-tier. Com a nova versão do DataSnap, muita coisa permaneceu igual, para permitir a compatibilidade (característica marcante do Delphi durante os seus mais de 15 anos de vida). Mas muita coisa também mudou, para podermos usufruir de novas tecnologias e tendências do mercado. Então nada melhor do que iniciar passo-a-passo, com um exemplo prático, no novo mundo multicamadas do Delphi 2010, como você vai conferir no artigo do Fabrício.

Esta edição ainda traz muitos artigos sobre boas práticas de desenvolvimento, engenharia de software, design patterns, refactoring, mapeamento objeto / relacional e outras maravilhas proporcionadas pela programação orientada a objetos. O Rafael fez algo muito, mas muito interessante, ele mostrou como desenvolver um software de forma errada, usando más práticas (espelho de muitas implementações que vemos por aí), para depois passo-a-passo indicar os próprios erros em pontos exatos, sugerindo melhorias tanto de design quanto implementação. Excelente técnica, tanto no artigo quanto no código. O Ricardo mostra a aplicação de padrões de projeto no mundo real. E no PHP não deixamos as boas práticas de lado, você verá como usar um robusto framework para mapeamento objeto / relacional.

Boa leitura!

Guinther Pauli

guinther@devmedia.com.br
Certificado Delphi 3,5,6,7,2005,2006,Web,Kylix
Microsoft Certified: MCP,MCAD,MCSD.NET,MCTS,MCPD (ASP.NET,C#)
guintherpauli.blogspot.com
twitter.com/guintherpauli

ClubeDelphi PLUS

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

DevMedia group

Perguntas e Respostas

ListView estilo Windows Vista / 7

Pergunta respondidas por:
Rodrigo Carreiro Mourão

Vejo muitos artigos sobre novas tecnologias nas novas versões do Delphi, porém as interfaces das aplicações estão evoluindo, o Win7 nos obriga a isso. Desde o Delphi 2009 uma série de novos componentes visuais foi adicionada e muitas propriedades novas foram criadas em componentes antigos, por exemplo o ListView. Como podemos criar um ListView estilo Windows Vista / 7?

Dúvida enviada pelo leitor Helimar Tavares - Rio de Janeiro

Sem dúvida alguma as interfaces das aplicações têm evoluído bastante junto com a maneira com que interagimos com as aplicações, Multitouch e Gestures são exemplos disso. Para os usuários Windows a principal referência é o sistema operacional, é comum as aplicações "imitarem" o sistema operacional. O grande problema é que nós migramos de versão do Delphi, mas não migramos os costumes, um exemplo disso é o componente TButton que desde o Delphi 2009 possui diversas novas funcionalidades, mas que pouca gente utiliza.

Quanto ao componente ListView acontece a mesma coisa. Como muitos controles Windows que temos no Delphi, ele evoluiu junto com o sistema operacional. Um exemplo de ListView no próprio sistema operacional é a janela Meu Computador do Windows Vista / 7. Veja na **Figura 1** que agora o ListView separa os ícones por grupo e este grupo pode ser manipulado. Agora podemos fazer isso de forma simples no Delphi. Vamos ver na prática.

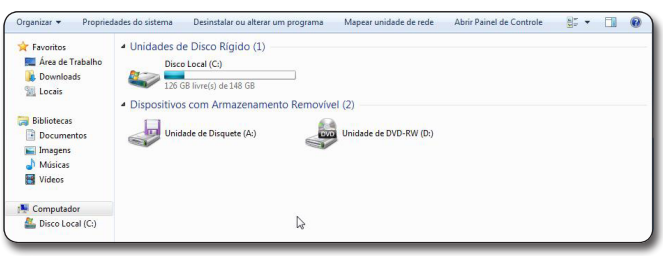


Figura 1. Exemplo de ListView agrupado

No Delphi 2010, acesse o menu File > New > VCL Forms Application e salve a aplicação. No form da aplicação adicione um componente ImageList e um ListView. No componente ImageList altere as propriedades Height e Width para 32, ou seja, este ImageList será para as imagens grandes que serão exibidas quando o ListView estiver configurado para exibir os itens como ícones.

Dê um duplo clique no ImageList e adicione algumas imagens para que possamos simular os itens do ListView. Você poderá encontrar alguns ícones em C:\Arquivos de programas\Arquivos comuns\CodeGear Shared\Images\GlyFX\Icons\XP\BMP\32x32. Lembre-se que este caminho pode variar dependendo do sistema operacional e versão do Delphi. Feito isso aponte a propriedade LargeImages do ListView para o ImageList1.

Como vamos trabalhar com os itens do ListView agrupados, vamos então criar dois grupos para que quando criarmos nossos itens já possamos associar a um dos grupos. Para isso, com o ListView selecionado, vá até a propriedade Groups e clique nas reticências. Isso irá invocar o editor de propriedades que utilizaremos para criar os grupos. Faça isso clicando no ícone Add New. A **Figura 2** mostra o editor de propriedades.

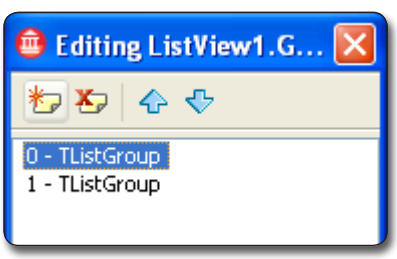


Figura 2. Criando os grupos

Selecione o primeiro grupo e altere a propriedade Header do mesmo para o texto que você quer que seja exibido no ListView, por exemplo, Contatos Pessoais ou então Discos Removíveis. Faça o mesmo para o segundo grupo e altere para o texto de sua preferência. Entre outras coisas podemos configurar uma imagem para o grupo, um texto de rodapé etc. Antes de finalizar selecione os dois grupos e expanda a propriedade state alterando a opção lgsCollapsible para true.

Com os grupos criados vamos então adicionar alguns itens no ListView. Para isso selecione o mesmo e clique na propriedade Itens. No diálogo que se abre informe um Caption para o item, associe a uma imagem através da opção ImageIndex e por fim selecione qual grupo este item vai pertencer através do ComboBox Group. A **Figura 3** mostra os dados sendo inseridos no ListView Items Editor.

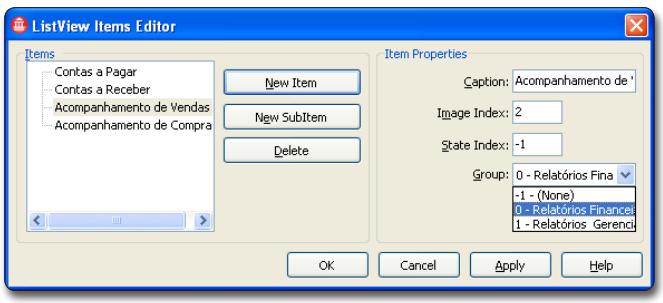


Figura 3. Criando os itens

Para concluir basta alterar a propriedade GroupView do ListView para true. Observe nas **Figuras 4** e **5** a aplicação em execução.

Envie suas dúvidas para nossa equipe de especialistas:
guinther@devmedia.com.br
pauloquicoli@devmedia.com.br
daniel.sobrinho@devmedia.com.br
rodrigo.carreiro@devmedia.com.br

Figura 4. ListView agrupado

Figura 5. ListView com grupo recolhido

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Do premiado autor da série
Dominando o Delphi: a Bíblia

Domine todos os novos recursos do Delphi, como o suporte a Unicode, os tipos genéricos, os métodos anônimos, os controles Ribbon (e outros novos controles), o DBExpress 4, o novo DataSnap e as novas funcionalidades do IDE.

Os novos recursos do Delphi são abordados em profundidade, com inúmeros exemplos de como são usados na prática. Os problemas que podem ser encontrados ao atualizar programas escritos em versões anteriores do Delphi (ao Delphi 7 ou Delphi 2007, por exemplo), em especial devido à introdução do suporte a Unicode, são discutidos em detalhes.

***Recursos Avançados do Delphi* é indispensável para todos os desenvolvedores, mesmo iniciantes, já que apresenta informações que não são encontradas em outros livros, e que são necessárias para escrever programas que façam proveito pleno dos novos recursos do Windows XP, Vista e 7.**

416 páginas – lançamento em 09/2009 – R$ 79,00

Nas melhores livrarias ou em
www.infopress.com.br

Quick Update: Autenticação no DataSnap 2010

Implemente segurança em aplicações multicamadas de uma forma simples

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br
Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Editor Técnico da Revista ClubeDelphi e Coordenador da área de Delphi no sistema de suporte on-line da DevMedia. Palestrante da Borland Conference 2007. Mantenedor da comunidade www.delphisophp.com, a primeira do Brasil 100% feita com Delphi for PHP.

A nova arquitetura DataSnap quebrou fronteiras no que diz respeito à integração de aplicações. O conceito de multicamadas com Delphi foi estendido. Até o Delphi 7 por exemplo a maneira mais simples de fazer uma interface Web para consumir um servidor de aplicação feito em Delphi Win32 era criar uma aplicação IntraWeb. Com o DataSnap no Delphi 2009 a barreira do COM para Win32 foi quebrada, adicionando transporte via TCP/IP, o que abriu novas possibilidades no lado cliente. O DataSnap 2009 tinha o framework dbExpress como base e podíamos agora utilizar Delphi Prism para criar clientes e consumir um servidor de aplicação feito em Delphi Win32.

Ainda tínhamos duas limitações. Primeiro a restrição de só trafegar tipos compatíveis com DBX, ou seja, ainda não poderíamos trafegar nossos objetos de negócio. Outra limitação é que, para criar uma Interface Web, ficávamos limitados à aplicações .NET feitas com Delphi Prism utilizando provider DBX. Qual o problema disso?

Quando pensamos em Delphi consumindo Delphi então não temos problemas, porém a realidade hoje nas empresas é outra. Cada dia que passa aumenta a necessidade de se

criar aplicações para integração entre plataformas. Um distribuidor de medicamentos e produtos farmacêuticos, por exemplo, pode ser impelido a permitir que seus clientes integrem seus sistemas com o ERP da empresa. Mas o ERP da empresa farmacêutica é em Delphi e seus clientes possuem PHP, ASP.NET - C#, Java etc. Como permitir isso de uma forma transparente, ou seja, aproveitando tudo o que tenho hoje, minhas regras, meus objetos, minhas rotinas? E mais, como fazer a comunicação entre cliente e servidor de forma otimizada? É neste contexto que o DataSnap 2010 entra.

Agora no DataSnap 2010 podemos trafegar dados utilizando HTTP como camada de transporte e com isso podemos criar aplicações REST tendo o JSON como "linguagem" de comunicação. Assim nossas regras, rotinas e objetos poderão ser consumidos via HTTP, por qualquer aplicação desde que a linguagem desta aplicação suporte JSON. Para ter noção você pode consumir um servidor utilizando apenas HTML e JavaScript.

Já que agora as fronteiras podem ser quebradas, então temos que ser mais cautelosos quanto à segurança. Neste Quick Update vamos aprender a adicionar autenticação via HTTP em um servidor DataSnap. Para montar nosso exemplo será necessário criar duas aplicações, uma que será nosso servidor e uma que seja nossa aplicação cliente. Para isso no Delphi 2010 selecione o Menu File > New > Other. Na caixa de diálogo que se abre busque por DataSanp. Selecione a opção DataSnapServer e clique em OK. Selecione as opções mostradas na **Figura 1**.

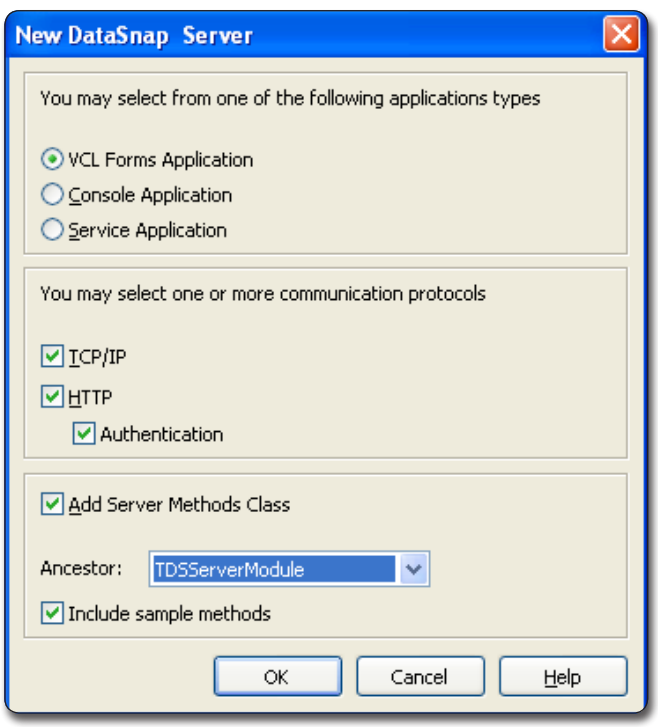


Figura 1. Criando o servidor

Em resumo estamos pedindo para criar um servidor DataSnap utilizando uma aplicação VCL com suporte a transporte via TCP/IP e HTTP e de quebra informamos que iremos trabalhar com autenticação. As últimas duas opções são para adicionar um Server Module e colocar um método de exemplo no mesmo.

Feito isso a aplicação será criada e uma série de componentes serão adicionados ao DataModule, chamado na aplicação de ServerContainer1. Salve a aplicação numa pasta Server em um diretório de sua preferência. De todos os componentes adicionados vamos dar atenção ao DSHTTPService1, que provê conexão ao servidor via HTTP, e o DSHTTPServiceAuthenticationManager1 que faz o trabalho de autenticação. Este último possui um evento HTTPAuthenticate que é invocado sempre que alguém tenta acessar um servidor. Observe a assinatura do evento na **Figura 2**.

Repare que o evento traz com ele todos os parâmetros necessários para fazermos a autenticação, neste caso, usuário e senha, além de um parâmetro por referência, booleano, para informar para o componente se a autenticação foi feita com sucesso ou não. Sem muita cerimônia, adicione no evento HTTPAuthenticate o código a seguir:

```
valid := (User = 'devmedia') and (Password = '123');
```

Obviamente que o código é apenas ilustrativo. Numa aplicação real estaríamos consultando uma base de dados para realizar a autenticação, mas mesmo assim dá para ter uma noção de como o método funciona. Se o usuário passado for devmedia e a senha 123, então nossa expressão retornará verdadeiro e passaremos isso para o parâmetro valid que se encarregará de permitir o acesso ao servidor.

Vamos ao cliente. Acesse File > New, porém agora selecionaremos a opção VCL Forms Application. Na aplicação criada apenas adicionaremos um componente SQLConnection. Na propriedade ConnectionName selecione DataSnapConnection que é a conexão de exemplo que vem com o Delphi. Uma vez feito isso clique no sinal de mais (+) ao lado da propriedade Driver. A **Figura 3** mostra as propriedades alteradas.

Repare que em CommunicationProtocol informamos HTTP, pois não há suporte à autenticação via TCP/IP, não

```
procedure TServerContainer1.DSHTTPServiceAuthenticationManager1HTTPAuthenticate(
  Sender: TObject; const Protocol, Context, User, Password: string;
  var valid: Boolean);
begin

end;
```

Figura 2. Evento de validação

com o HTTPAuthenticationManager. Além desta mudamos também a porta para 80, para que o SQLConnection utilize esta porta para conexão. Por fim alteramos LoginPrompt para false. Antes de executar a aplicação, vamos dar uses na unit DSHTTPLayer, necessária quando se trabalha com autenticação HTTP. Para testar adicione um botão na aplicação e no clique deste botão coloque o código a seguir:

```
DevServer.Open;
```

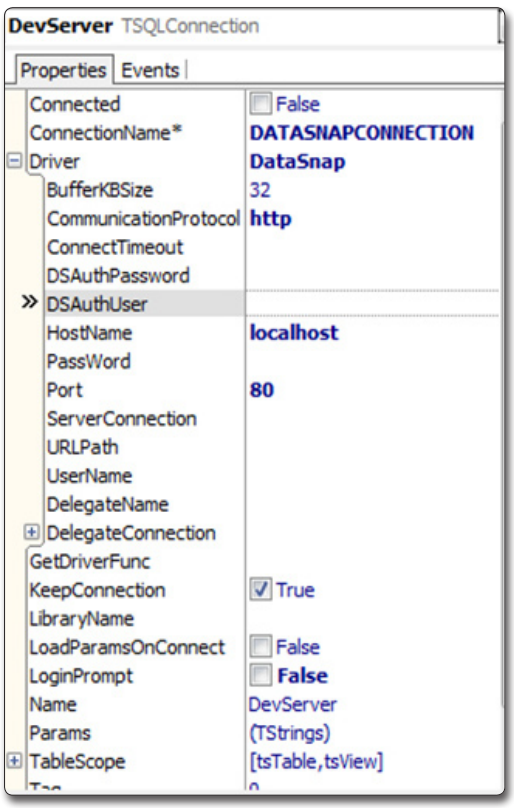


Figura 3. Configurando o SQLConnection

Com a aplicação servidora rodando execute o cliente e clique no botão. A **Figura 4** exibe a exceção gerada.

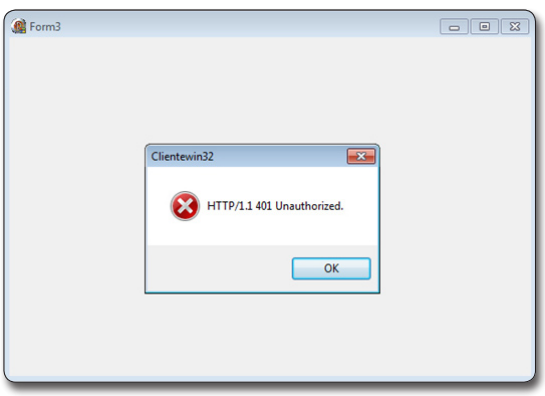


Figura 4. Exceção gerada

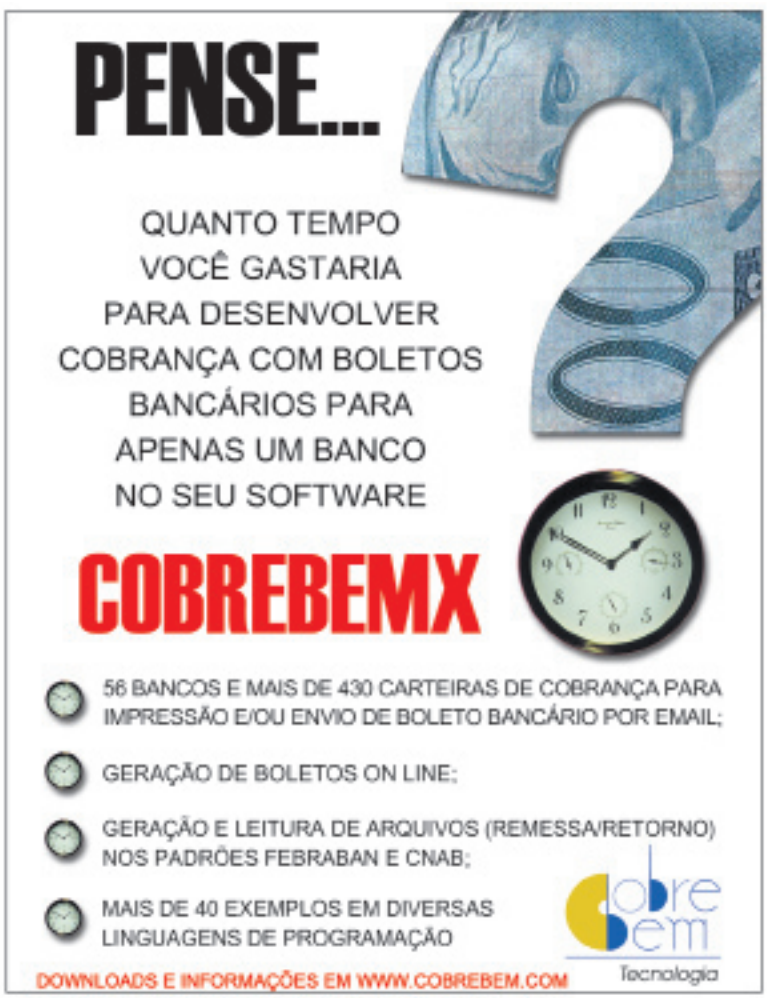
A exceção ocorre pois não informamos o usuário e a senha para autenticação. Volte ao Object Inspector e configure DSAuthUser como devmedia e DSAuthPassword como 123. Execute novamente a aplicação e clique no botão. Observe que não houve exceção, o que significa que estamos conectados no servidor com sucesso. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Mandamentos da Orientação a Objetos

Conheça os bons princípios que podem garantir qualidade

Paulo Quicoli
pauloquicoli@gmail.com
Editor técnico das revistas ClubeDelphi, .NET Magazine e WebMobile Magazine. Formado em processamento de dados pela FATEC-TQ. Atua como analista e desenvolvedor da ControlM Informática (www.controlm.com.br) e professor de linguagem de programação do curso técnico em processamento de dados do SENAC/Jaboticabal. Utiliza Delphi desde sua primeira versão, entusiasta do desenvolvimento orientado a objetos, tem publicado vários artigos sobre o assunto. Blog: www.pauloquicoli.spaces.live.com

Resumo do DevMan

Os princípios de modelagem de objetos são guias que nos levam a construir softwares flexíveis e estáveis. Evitando dependência excessiva entre as classes, duplicação de código e outros problemas mais. Neste artigo vamos estudá-los com exemplos práticos para que você leitor os assimile e possa utilizá-los no seu dia-a-dia.

Sempre se fala da orientação a objetos como a bala de prata para todos os problemas, desde análise à programação. Porém, colocar tudo em classes não resolve o problema, podendo até mesmo criar muito mais problemas, pois softwares orientados a objetos mal construídos são sinônimos de dores de cabeça.

Do que trata o artigo

Desenvolver sistemas é mais do que criar classes ao esmo, formulários e colocar tudo ali. É pensar no problema, possíveis soluções e unir tudo em algo que não cause mais problemas. Por isso existem os bons princípios para desenvolvimento de softwares orientados a objetos. Neste artigo vamos estudar os mais divulgados.

Para que serve

Os princípios são resultado de estudos da engenharia de software, e são utilizados para que se criem sistemas que sejam fáceis de dar manutenção, que possam responder rápido às mudanças de regra de negócios e flexíveis.

Em que situação o tema é útil

Desenvolver um sistema flexível é bom para você, como desenvolvedor, e bom para seu cliente. Para você é bom porque levará menos tempo para ajustar o sistema e o ajustará com segurança. Para seu cliente é bom porque ele verá que o sistema que ele utiliza é estável e que não vai deixá-lo na mão.

Acredito que todos os leitores já sabem o que é uma classe, uma propriedade, um método. Já conhecem estruturalmente a

orientação a objetos, o que falta é saber como modelar tudo isso de forma adequada.

Regras de negócio mudam constantemente e isso faz com que seja um desafio construir um software que responda rápido às mudanças apresentadas. Aplicativos bem desenvolvidos podem oferecer vantagem competitiva para seus clientes, contudo o projeto mal modelado pode levar a prejuízos por enrijecer processos, já que não se adapta às mudanças de forma eficiente. Vamos identificar os elementos de um bom design e de um não tão bom. Enquanto estiver lendo esses elementos, compare com o(s) sistema(s) que você desenvolve(u).

Modelagem boa

Aplicações bem desenhadas oferecem rotinas que são robustas, de fácil manutenção e reutilizáveis. Elas devem estar aptas a se adequar às mudanças sem afetar sua modelagem. Um exemplo disso seria uma aplicação onde é necessário exportar um arquivo em um determinado formato. Adicionar novos formatos ao sistema atual deve ser uma tarefa fácil. As três características de um bom design são:

• **Manutenção** – É a facilidade com que um sistema pode ser alterado para aplicar novas regras de negócio exigidas, mas não só regras e sim também melhorar seu desempenho, corrigir falhas e outras solicitações mais. Aplicações que possuem um bom design exigem poucos recursos (tempo, esforço e até mesmo capital) para realizar sua devida manutenção;

• **Reusabilidade** – Em um sistema adequado é possível que os seus componentes possam ser reutilizados por outros sem grandes dificuldades;

• **Robustez -** É a estabilidade do sistema em situações extremas do tipo: tratamento de entradas erradas por parte de usuários, carga máxima de dados em determinado momento do dia.

Modelagem ruim

Ninguém desenvolve um software não muito bom porque quer. Isso é decorrência da inexperiência ou de uma modelagem feita às pressas para atender a um tempo limite de entrega. Sistemas construídos dessa maneira costumam ter os seguintes problemas em comum:

• **Rigidez** – Dizemos que um sistema é rígido quando uma alteração nele é difícil. E essa alteração é considerada difícil porque rotinas internas estão, muito provavelmente, entrelaçadas a um ponto onde alterar qualquer coisa leva a uma cascata de alterações. Nessas situações quando o sistema fica grande, é difícil estimar o prazo para uma alteração já que todo o impacto deve ser medido antes;

• **Fragilidade** – Por ter rotinas muito ligadas umas as outras, a alteração em uma leva a problemas nas outras. Quem aqui nunca alterou algo em um sistema que depois causou um problema em uma parte que parecia não estar relacionada com a alteração feita? Corrigir esse tipo de problema pode levar a outros, desencadeando uma leva de ajustes que custam tempo e que consequentemente custam dinheiro. Um sistema nessa situação perde sua credibilidade, pois vive em contenção de erros, impedindo que seus desenvolvedores mantenham uma qualidade futura ao projeto;

• **Reusabilidade** – Neste caso, a falta da mesma. Quando for preciso reutilizar alguma rotina, o trabalho pode ser tão difícil que é menos custoso desenvolver novamente essa rotina.

Agora que vimos como se parece, do ponto de vista arquitetural, um bom sistema e um ruim, vamos conhecer alguns princípios que podem determinar o sucesso de design.

Seja S.O.L.I.D.

Essa sigla é um acrônimo para a palavra em inglês Solid, que significa sólido, solidez. Cada letra representa um acrônimo de um princípio, ou seja, temos um acrônimo de um acrônimo. Esse conjunto é o resultado de estudos e experiências de vários desenvolvedores, e foi primeiramente catalogado por Robert "Uncle Bob" Martin (veja sessão links) em seu livro Applying Principles and Patterns. Vamos conhecer cada princípio.

S de Single Responsability Principle (SRP)

Este é o princípio da responsabilidade única, que recomenda que uma classe só deva ter um motivo para sofrer alguma alteração. Sendo assim, podemos resumir que uma classe qualquer em seu sistema deva ter uma tarefa única e específica. Isso contudo não quer dizer que se tenha apenas um único método, mas que todos os métodos criados trabalhem juntos em um único objetivo, atender a responsabilidade da classe. Outros autores também chamam este princípio de coesão.

Para entendermos seu significado vamos imaginar um cenário que todos conhecem. Digamos que você possui uma classe que represente um Nota Fiscal, a classe TNotaFiscal. E você a codificou de tal forma que ela mesma, além de conter seus itens, seus cálculos, também sabe como se salvar e se recuperar do banco de dados e até mesmo sabe como se exportar nos mais diversos formatos. Se pararmos para pensar, você acha que é responsabilidade da TNotaFiscal saber tudo isso? Apesar da forma que foi construída, temos essa classe lidando com mais de uma situação de responsabilidades diferentes. Se amanhã a forma de exportação é alterada, é preciso corrigir a classe. Se também algo muda na regra de negócio referente a uma nota fiscal, essa classe também precisa ser corrigida, ou seja, temos aqui mais de um motivo para essa classe ser alterada, e isso não é bom. Não é bom porque cada responsabilidade é um arco de mudanças, que muito provavelmente estão entrelaçadas. Estamos falando aqui de responsabilidade, mas o que é isso para uma classe?

No contexto deste artigo, ela representa um motivo de mudança. Nem sempre é fácil identificar responsabilidades a mais em uma classe, isso porque estamos acostumados a ver como um todo. Por exemplo, veja a classe TCliente da **Listagem 1**.

Esse é um design que já vi muito por aí. Observe e veja que, além de ter a responsabilidade de manter os dados dos clientes, é responsabilidade da classe TCliente se salvar, excluir e se localizar em algum banco de dados. Estamos acoplando, adicionando responsabilidades a mais, que poderiam inviabilizar a reutilização de TCliente. Na **Listagem 2** temos uma outra proposta, que respeita o princípio SRP.

Listagem 1. Classe TCliente

```
TCliente = class
 private
  FDataNascimento: TDateTime;
  FId: integer;
  FNome: string;
  procedure SetDataNascimento(const Value: TDateTime);
  procedure SetId(const Value: integer);
  procedure SetNome(const Value: string);
 public
  property Id: integer read FId write SetId;
  property Nome: string read FNome write SetNome;
  property DataNascimento: TDateTime read FDataNascimento write SetDataNascimento;
  procedure Salvar(cliente: TCliente);
  procedure Excluir(cliente: TCliente);
  function Localizar(nomeCliente: string) : TList<TCliente>;
end;
```

Listagem 2. Aplicando SRP

```
TClienteSRP = class;

IDaoClienteSRP = interface
 ['{FC3651D9-3B1F-459F-B1C4-6EAA4616E0BC}']
  procedure Salvar(cliente: TClienteSRP);
  procedure Excluir(cliente: TClienteSRP);
  function Localizar(nomeCliente: string): TList<TClienteSRP>;
end;

TClienteSRP = class
 private
  FDataNascimento: TDateTime;
  FId: integer;
  FDao: IDaoClienteSRP;
  FNome: string;
  procedure SetDao(const Value: IDaoClienteSRP);
  procedure SetDataNascimento(const Value: TDateTime);
  procedure SetId(const Value: integer);
  procedure SetNome(const Value: string);
 published
 public
  property Id: integer read FId write SetId;
  property Nome: string read FNome write SetNome;
  property DataNascimento: TDateTime read FDataNascimento write SetDataNascimento;
  property Dao: IDaoClienteSRP read FDao write SetDao;
  constructor Create(dao: IDaoClienteSRP); reintroduce;
end;
```

Este design agora sim é superior ao anterior, porque separamos as funções de acesso a banco de dados em outra classe, representada pela interface IDaoClienteSRP. Mesmo estando em uma classe separada, nós ligamos a classe TCliente à IDaoClienteSRP através uma propriedade chamada Dao, embora fazer isso não seja o ideal, em alguns casos pode ser necessário. Este princípio é o mais simples, porém é o mais difícil de ser feito certo, pois exige boa interpretação, identificação e bom senso do desenvolvedor.

O de Open/Closed Principle (OCP)

Este princípio é muito importante para se obter um bom design de software. Ele determina que suas classes (classes de negócio, serviços etc.) devem ser abertas para extensão e fechadas para modificação. A ideia aqui é que você nunca altere uma classe que esteja funcionando (salvo mudanças de requisitos nela própria), mas sim, que possa estendê-la para ser reutilizada em outros cenários. Isto para evitar que bugs sejam adicionados ao que já está funcionando. Vamos ver um exemplo prático disso.

Imagine que em um sistema seja necessário processar alguns pedidos de compra. Para isso então você desenvolve uma classe TPedido e uma classe TValidadorPedido, conforme **Listagem 3**.

Listagem 3. Classes de negócio

```
TPedido = class
 private
  FCodProduto: integer;
  FQuantidade: integer;
  FData: TDateTime;
  procedure SetCodProduto(const Value: integer);
  procedure SetData(const Value: TDateTime);
  procedure SetQuantidade(const Value: integer);
 public
  property Data: TDateTime read FData write SetData;
  property CodProduto: integer read FCodProduto write SetCodProduto;
  property Quantidade: integer read FQuantidade write SetQuantidade;
 end;

 TValidadorPedido = class
 private
  FEstaValido: boolean;
  FErros: string;
  procedure SetErros(const Value: string);
  procedure SetEstaValido(const Value: boolean);
 public
  property Erros: string read FErros write SetErros;
  property EstaValido: boolean read FEstaValido write SetEstaValido;
  procedure Validar(pedido: TPedido);
 end;
```

Nota

As classes e suas implementações são simples e servem apenas para ilustrar o caso, em uma situação real mais classes estariam envolvidas.

A classe TValidadorPedido é responsável por validar um pedido, indicando que o mesmo pode ser processado ou não. Por isso nela temos um método Validar. Nele estarão todas as validações que um pedido, de acordo com a regra de negócio, pode sofrer. Caso algo não esteja correto, isso é registrado na propriedade Erros e o estado do pedido é então armazenado na propriedade EstaValido. Veja na **Listagem 4** como foi implementado o método Validar(pedido: TPedido).

Nota do DevMan

Pensando nas camadas de um sistema: Apresentação, Regras de Negócio e Acesso a Dados, um modelo DAO (Data Access Object) são classes localizadas na camada de acesso a dados e responsáveis por recuperar, salvar e excluir objetos. Caso você tenha um objeto de negócio chamado Aluno, então para ele você terá uma classe AlunoDAO que terá métodos para procurar um aluno, salvar ou excluir o mesmo de um meio físico. Dessa forma estamos separando das regras de negócio o acesso ao banco de dados, mantendo cada coisa em seu lugar.

Listagem 4. Validando um pedido

```
procedure TValidadorPedido.Validar(pedido: TPedido);
begin
 Erros := '';
 EstaValido := true;
 if DaysBetween(pedido.Data,now) > 15 then
 begin
  Erros := Erros + 'Pedido anterior a 15 dias';
  EstaValido := false;
 end;
end;
```

Aqui para simplificar temos apenas uma validação que verifica se o pedido tem mais de 15 dias. Se tiver ele é considerado inválido. Mas imagine a situação real, onde vários cenários devem ser verificados para se aprovar ou não um pedido e que todas essas verificações estão aí, dentro desse método Validar. Com certeza ele será bem cumprido. Agora imagine que a regra de negócio mudou e você precisa adicionar mais uma validação. Como implementamos isso na situação atual? Muito bem, a solução aqui é alterar a classe TValidarPedido e ajustar o método Validar para incluir dentro dele a nova validação exigida.

Isso parece muito normal, mas fere o conceito OCP. Veja que estamos alterando a classe e não a estendendo. Digamos que nesse sistema surgiu uma nova categoria de pedidos que também precisam ser validados, como você reaproveitaria essa classe? Estamos fazendo ajustes o que não é recomendado. Alterar esse método que já funciona pode introduzir algum bug não esperado. Vamos então ajustar nosso design para que fique de acordo com o princípio, e assim, possamos reutilizar a classe.

A primeira coisa que vou fazer é tirar de dentro do método Validar os códigos de validação. Cada regra de validação será uma classe e TValidadorPedido terá uma lista dessas regras. Então, o que o método Validar terá que fazer, é percorrer essa lista executando cada regra. Veja que basicamente essa validação deve fazer uma consistência e retornar um texto com a mensagem de erro. Sendo assim defini a seguinte interface que será implementada pelas regras:

```
IValidacaoPedido = interface
['{20FDABD0-091C-4C97-8D16-E12F7D964DC9}']
  function Validar(pedido: TPedido): string;
end;
```

Agora digamos que será preciso continuar com a validação da data do pedido. Logo, implementamos uma classe para isso, como mostra a **Listagem 5**. Agora veja na **Listagem 6** como fica a classe TValidadorPedido.

Observe que o método Validar agora percorre sua lista de regras, executando uma a uma. Com isso não é mais o método Validar que realmente faz a verificação e sim as regras contidas nele. Para usá-lo basta passar essa regra, como mostra o código a seguir:

```
Obj := TValidadorPedido.Create;
Obj.ListaValidacao.Add(TValidaDataEmissao.Create);
Obj.Validar(objPedido);
```

Listagem 5. Validando a data de emissão

```
TValidaDataEmissao = class(TInterfacedObject, IValidacaoPedido)
 public
  function Validar(pedido: TPedido): string;
 end;

function TValidaDataEmissao.Validar(pedido: TPedido): string;
var
 erro: string;
begin
 erro:= '';
 if DaysBetween(pedido.Data, now) > 15 then
  erro:= 'Pedido anterior a 15 dias';
 result := erro;
end;
```

Listagem 6. TValidadorPedido utilizando OCP

```
TValidadorPedidoOCP = class
private
 FEstaValido: boolean;
 FErros: string;
 FListaValidacao: TList<IValidacaoPedido>;
 procedure SetErros(const Value: string);
 procedure SetEstaValido(const Value: boolean);
 procedure SetListaValidacao(const Value: TList<IValidacaoPedido>);
published
public
 constructor Create; reintroduce;
 destructor Destroy; override;
 property Erros: string read FErros write SetErros;
 property EstaValido: boolean read FEstaValido write SetEstaValido;
      property ListaValidacao: TList<IValidacaoPedido> read FListaValidacao write
      SetListaValidacao;
 procedure Validar(pedido: TPedido);
end;

procedure TValidadorPedidoOCP.Validar(pedido: TPedido);
var
 validacao: IValidacaoPedido;
begin
 Erros := '';
 for validacao in FListaValidacao do
  Erros := Erros + validacao.Validar(pedido);
 EstaValido := Trim(Erros)='';
end;
```

Digamos que agora as regras mudaram e mais uma validação deve ser feita, agora será preciso validar a quantidade do produto requerido. O que fazemos? Vamos alterar TValidadorPedido? Não, vamos implementar a nova regra e adicioná-la na lista, como mostra **Listagem 7**.

Veja que criamos a classe TValidaQuantidade que implementa nossa interface IValidacaoPedido. Então no método Validar codificamos o que é necessário, lembrando que para efeito de simplificação, o método é resumido ao máximo. Na sequência vemos como adicionamos isso a TValidadorPedido:

```
Obj := TValidadorPedido.Create;
Obj.ListaValidacao.Add(TValidaDataEmissao.Create);
Obj.ListaValidacao.Add(TValidaQuantidade.Create);
Obj.Validar(objPedido);
```

Posso citar algumas vantagens dessa nova implementação em relação à anterior:

• Não há necessidade de se alterar a classe TValidador Pedido;
• Fácil inclusão/remoção de regras;
• TValidadorPedido agora pode ser reutilizado, pois agora ele valida conforme regras adicionadas a ele, ou seja, se tornou flexível;

Mas também vejo um contra em tudo isso. Como cada nova regra de validação será uma classe, haverá um aumento de classes criadas e a lógica pode ficar um pouco dispersada, o que leva iniciantes a ficarem um pouco confusos ao lidar com esse tipo de estrutura.

L de Liskov Substitution Principle (LSP)

Este princípio diz que, se você possui um método ou função que utiliza outra classe, caso você passe para esse método uma subclasse, esse método deveria continuar a funcionar perfeitamente. Por exemplo, você possui o método TImpressao. Imprimir que deve listar os dados da classe TPessoa, como mostra a **Listagem 8**.

Veja que temos a classe TPessoa, que possui apenas um atributo, Nome. Ao passarmos para o método Imprimir uma instância de TPessoa, ele irá imprimir a propriedade Nome. Agora, digamos que nossa regra de negócio mudou e temos uma subclasse de TPessoa, temos TPessoaJuridica, como é mostrado a seguir:

```
TPessoaJurica = class(TPessoa)
 private
  FRazaoSocial: string;
  procedure SetRazaoSocial(const Value: string);
 public
  property RazaoSocial: string read FRazaoSocial write SetRazaoSocial;
end;
```

Se você passar agora uma instância de TPessoaJuridica para o método TImpressao.Imprimir ele irá funcionar? Sim, mas o resultado será o esperado? Lembre-se, o objetivo desse método é listar todos os dados da pessoa passada no parâmetro. Ou seja, o resultado não será o esperado porque que Imprimir irá listar apenas o nome. É justamente disso que o princípio LSP trata. Se você quer utilizar uma hierarquia de classes, e possui um método que utiliza um item dessa hierarquia, caso você passe para esse método uma subclasse, ele deve continuar a funcionar como o esperado. Da maneira que temos nossas classes aqui, elas violam esse princípio. Vamos adequá-las.

A primeira coisa que vou fazer é criar um método em TPessoa que retorne em uma lista os dados de suas propriedades, veja na **Listagem 9**.

Como pode ser visto, TPessoaLSP define o método Retorna Dados que devolve uma variável TStrings que contém, na forma de lista, os dados da instância. Observe que esse método está marcado como virtual, assim nós podemos sobrecarregá-lo para que seja usado por subclasses de TPessoaLSP. No caso, vamos utilizá-lo na subclasse TPessoaJuridicaLSP (**Listagem 10**).

Listagem 7. Criando a nova regra e adicionando em TValidadorPedido

```
TValidaQuantidade = class(TInterfacedObject, IValidacaoPedido)
public
 function Validar(pedido: TPedido): string;
end;

function TValidaQuantidade.Validar(pedido: TPedido): string;
begin
 if (pedido.CodProduto = 2) and (pedido.Quantidade>5) then
   result := 'O produto 2 só pode ser vendido em até 5 unidades por pedido'
 else
  result := '';
end;
```

Listagem 8.Imprimindo TPessoa

```
TPessoa = class
 private
  FNome: string;
  procedure SetNome(const Value: string);
 public
 property Nome: string read FNome write SetNome;
end;

TImpressao = class
 public
  procedure Imprimir(pessoa: TPessoa);
end;

procedure TImpressao.Imprimir(pessoa: TPessoa);
begin
 WriteIn(pessoa.Nome);
end;
```

Listagem 9. Ajustando TPessoa ao LSP

```
TPessoaLSP = class
 private
  FNome: string;
  procedure SetNome(const Value: string);
 public
  function RetornaDados: TStrings; virtual;
  property Nome : string read FNome write SetNome;
end;

function TPessoaLSP.RetornaDados: TStrings;
var
 resultado: TStrings;
begin
 resultado := TStrings.Create;
 resultado.Add(self.Nome);
 result := resultado;
end;
```

Listagem 10. TPessoaJuridicaLSP

```
TPessoaJuridicaLSP = class(TPessoaLSP)
 private
  FRazaoSocial: string;
  procedure SetRazaoSocial(const Value: string);
 public
  function RetornaDados: TStrings; override;
  property RazaoSocial: string read FRazaoSocial write SetRazaoSocial;
end

function TPessoaJuridicaLSP.RetornaDados: TStrings;
var
 resultado: TStrings;
begin
 resultado := TStrings.Create;
 resultado := inherited RetornaDados;
 resultado.Add(Self.RazaoSocial);
 result := resultado;
end;
```

Como podemos ver TPessoaJuridicaLSP sobrecarrega o método RetornaDados (override) para que em sua instância ele retorne também dados que só ela possui, como Razao-Social por exemplo. Mas observe como foi implementado RetornaDados, estamos invocando a execução da classe base em:

```
resultado := inherited RetornaDados;
```

Dessa forma o nome será incluído automaticamente. Agora é preciso ajustar o método de impressão. Veja como ele é alterado, na **Listagem 11**.

Veja que agora o método Imprimir faz uso do método Retorna Dados definido em TPessoaLSP. Agora sim, se passarmos qualquer descendente de TPessoaLSP, que sobrecarregue o método RetornaDados para incluir suas peculiaridades, todos os dados serão listados. Com isso, podemos ver que manter uma hierarquia de classes não é tão simples e até mesmo seu uso extensivo é desencorajado.

I de Interface Segregation Principle (ISP)

Este princípio está relacionado ao primeiro, SRP. Aqui é dito que não devemos obrigar uma classe a implementar métodos de uma interface que ela não necessita. Isso cabe também a classes abstratas. Vamos imaginar que estamos estruturando nossos objetos de negócio, e para isso vamos criar uma interface que será implementada por uma classe base que será herdada por todos os objetos de negócio, como mostra a **Listagem 12**.

Veja que a interface IObjetoNegocio expõe uma série de métodos. Por exemplo, BeginEdit, CancelEdit e EndEdit são utilizados na edição dos objetos de negócio. O Validar será utilizado por objetos que precisam ser validados. E por fim temos o RetornaDados, que deve retornar os valores das propriedades. Então temos uma classe base, TObjetoNegocio, que fará com que todas nossas classes tenham essas funcionalidades, mesmo não precisando. Este é o ponto deste princípio. Fazer suas classes terem coisas que não precisam. Digamos que você tenha um objeto de negócio que não precisa ser validado, porque ele deve implementar validação então?

A solução está em criar interfaces bem específicas, assim, você escolhe o que implementar e o que não implementar. Vamos então ajustar no exemplo organizando primeiramente a interface IObjetoNegocio. Ela está muito cheia de funcionalidades. Veja na **Listagem 13** como ela pode ficar.

Como pode ser visto, no lugar de uma, agora temos quatro interfaces, todas representando funcionalidades específicas que podem ser escolhidas quando for necessário. Mas ainda não acabamos, vamos ajustar as classes como mostra a **Listagem 14.**

Veja que agora nós podemos especializar nossos objetos de negócio, se uma classe não vai ser validada, não precisa mais implementar métodos necessários, basta agora herdar da classe base adequada.

D de Dependency Inversion Principle (DIP)

A definição formal desse princípio diz que módulos de alto nível não podem depender de módulos inferiores, mas devem depender de abstrações. E que abstrações não devem depender de detalhes, mas os detalhes é que devem depender das abstrações.

Vamos colocar tudo isso em uma linguagem mais simples e objetiva: quando suas classes precisam de referências de

Listagem 11. Classe TImpressaoLSP

```
TImpressaoLSP = class
 public
  procedure Imprimir(pessoa: TPessoaLSP);
end;

procedure TImpressaoLSP.Imprimir(pessoa: TPessoaLSP);
var
 s: string;
begin
 for s in pessoa.RetornaDados do
  WriteLn(s);
end;
```

Listagem 12. Interface e classe base de negócio

```
IObjetoNegocio = interface
 ['{3DC8871F-6686-41A1-A058-D27E7FED9B31}']
 property Id: integer;
 procedure BeginEdit;
 procedure CancelEdit;
 procedure EndEdit;
 procedure Validar;
 function RetornaDados:TStrings;
end;

TObjetoNegocio = class(TInterfacedObject, IObjetoNegocio)
private
 FId: integer;
 procedure SetId(const Value: integer);
public
 property Id: integer read FId write SetId;
 procedure BeginEdit;
 procedure CancelEdit;
 procedure EndEdit;
 procedure Validar;
 function RetornaDados : TStrings;
end;
```

Listagem 13. Interfaces organizadas

```
IEntidadeISP = interface
 ['{C0929547-67AA-4901-8FAE-BF95071A7C43}']
 property Id: integer;
end;

IEditavelISP = interface
 ['{9EA5CE89-A571-4D60-A2FD-9C80CC650CB5}']
 procedure BeginEdit;
 procedure CancelEdit;
 procedure EndEdit;
end;

IValidavelISP = interface
 ['{62A98938-F852-4A8E-8B0E-E7C3B51279FC}']
 procedure Validar;
end;

IRetornavelISP = interface
 ['{741A19E2-35D1-4012-94DB-98B2C4B9D2C6}']
 function RetornaDados: TStrings;
end;
```

outras classes, que essas referências sejam interfaces. A ideia é isolar nossa classe atrás de uma "cerca" bem definida. Essa cerca é a abstração na qual a classe depende. Estando presa a uma abstração e não a algo concreto, caso a implementação da abstração sofra alterações, a classe não sofrerá nada. Mais uma vez vamos expor esse princípio na prática. Imagine que seu aplicativo precise exportar um objeto inicialmente para o formato texto. Assim temos a seguinte classe da **Listagem 15**.

Veja que a classe TExportador faz algo concreto, diretamente. Está ligada ao conceito de exportar no formato texto o objeto passado como parâmetro. Caso amanhã um novo formato necessite de ser criado, por exemplo, a XML? Talvez você ajustasse a classe como a seguir:

```
TExportador = class
public
  procedure ExportaProdutoEmTexto(obj: TProduto);
  procedure ExportaProdutoEmXML(obj: TProduto);
end;
```

Entenda que TExportador ainda está ligado a um conceito concreto, exportar em XML. TExportador não é muito flexível porque a cada novo formato temos que alterar sua estrutura adicionando mais métodos. TExportador deve ser independente dos detalhes que controla, é sobre isso este princípio. Vamos aplicar isso nele então. Veja na **Listagem 16** como deve ficar a classe.

Foi definida uma interface que representa a abstração mencionada anteriormente, temos o conceito IExportar agora. É nele que nosso exportador, TExportadorDIP, irá depender. Como IExportar é apenas uma abstração é preciso criar a classe que a implementa, nesse caso temos TFormatoXML e TFormatoTexto. Para executar o exportador é muito simples agora:

```
Exporta := TExportadorDIP.Create;
Exporta.Exportar(produto, TFormatoXML.Create);
```

Se for necessário mudar o formato de exportação ou criar um novo, apenas criamos esse formato e o passamos para TExportador:

```
Exporta.Exportar(produto, TFormatoTexto.Create);
```

Se olharmos a implementação do método TExportadorDIP.Exportar veremos que para TExportadorDIP não interessa qual o formato da exportação, ele sabe apenas que existe um método chamado Exportar, que foi definido pela abstração IExportar.

Outros princípios importantes

Existem outros princípios, que não foram catalogados formalmente, mas compõem um conjunto, informal, de boas práticas a serem seguidas.

Listagem 14. Classes organizadas

```
TEntidadeISP = class(TInterfacedObject, IEntidadeISP)
private
  FId: integer;
  procedure SetId(const Value: integer);
published
public
  property Id: integer read FId write SetId;
end;

TEntidadeValidavelISP = class(TEntidadeISP, IValidavelISP)
public
  procedure Validar;
end;

TEntidadeValidavelEditavelISP = class(TEntidadeValidavelISP, IEditavelISP)
public
  procedure BeginEdit;
  procedure CancelEdit;
  procedure EndEdit;
end;

TEntidadeCompletaISP = class (TEntidadeValidavelEditavelISP, IRetornavelISP)
public
  function RetornaDados: TStrings;
end;
```

Listagem 15. Classes sem DIP

```
TProduto = class
private
  FNome: string;
  procedure SetNome(const Value: string);
public
  property Nome: string read FNome write SetNome;
end;

TExportador = class
public
  procedure ExportaProdutoEmTexto(obj: TProduto);
end;
```

Listagem 16. TExportador aplicando DIP

```
IExportar = interface
  ['{5520CFA2-4970-4414-B851-B08377DDD5BE}']
  procedure Exportar(obj: TProduto);
end;

TFormatoTexto = class(TInterfacedObject, IExportar)
public
  procedure Exportar(obj: TProduto);
end;

TFormatoXML = class(TInterfacedObject, IExportar)
public
  procedure Exportar(obj: TProduto);
end;

TExportadorDIP = class
public
  procedure Exportar(obj: TProduto; Formato: IExportar);
end;
```

Law of Demeter

Este princípio diz que uma determinada classe C, que possui um método M, esse método só pode invocar outros métodos que pertencem:

- À classe C;
- Membros de C;
- Parâmetros de M;

• Objetos criados por M;
• Objetos criados por outros métodos da classe C chamados por M;
• Objetos Globais.

Este princípio é bom porque tende a diminuir o relacionamento entre as mais diversas classes e com isso o efeito cascata de uma alteração é reduzido. Vejamos um exemplo prático. Digamos que temos uma classe TReajuste que serve para reajustar o valor de promoção de produtos e consequentemente ajustar seu valor. Pensando nesse cenário temos as seguintes classes, veja **Listagem 17**.

Listagem 17. Cenário que não aplica a lei de Demeter

```
TPromocao = class
private
  FDesconto: currency;
  procedure SetDesconto(const Value: currency);
public
  property Desconto: currency read FDesconto write SetDesconto;
end;

TProduto = class
private
  FValor: currency;
  FPromocao: TPromocao;
  FNome: string;
  procedure SetNome(const Value: string);
  procedure SetPromocao(const Value: TPromocao);
  procedure SetValor(const Value: currency);
public
  property Nome: string read FNome write SetNome;
  property Valor: currency read FValor write SetValor;
  property Promocao: TPromocao read FPromocao write SetPromocao;
end;

TReajuste = class
private
  FProdutos: TList<TProduto>;
  FTotalDesconto: currency;
  procedure SetProdutos(const Value: TList<TProduto>);
  procedure SetTotalDesconto(const Value: currency);
public
  constructor Create;reintroduce;
  property Produtos: TList<TProduto> read FProdutos write SetProdutos;
  property TotalDesconto: currency read FTotalDesconto write SetTotalDesconto;
  procedure ReajustarPromocao;
end;
```

Temos a classe TPromocao que retém o percentual de desconto. Temos também as classe produto que possui uma referência a TPromocao e por fim TReajuste, que possui uma lista de produtos, um método que reajusta a promoção e o valor total que foi descontado de cada produto. Vamos ver agora como o método ReajustarPromocao funciona (**Listagem 18**).

Vamos entender porque esse método viola a lei de Demeter. Observe que o método acessa a propriedade Promocao, que é uma referência a TPromocao, de cada produto em sua lista. Isso não é certo porque o objeto Promocao não é membro da classe TReajuste, não é passado como parâmetro do método ReajustarPromocao, não é instanciado dentro do método ReajustarPromocao e não é um objeto global. Acessá-lo dessa maneira faz que nossa classe TReajuste esteja ligada não só a TProduto, mas agora a TPromocao e caso esta venha sofrer alguma alteração, talvez tenhamos que ajustar também TReajuste, isso é o chamado alto acoplamento, indesejável em um sistema orientado a objetos.

Mas, vamos corrigir tudo isso para diminuir o acoplamento e empregar a lei de Demeter. O resultado você vê na **Listagem 19**.

Listagem 18. Método ReajustarPromocao

```
procedure TReajuste.ReajustarPromocao;
var
 obj: TProduto;
begin
 TotalDesconto := 0;
 for obj in Produtos do
 begin
  if obj.Valor <= 10 then
  begin
   obj.Promocao.Desconto := 2;
   obj.Valor := obj.Valor - ((2 * obj.FValor)/100);
   TotalDesconto := TotalDesconto +((2 * obj.FValor)/100);
  end;
 end;
end;
```

Listagem 19. Refatoração para aplicar Lei de Demeter

```
function TProduto.FazerReajuste(porcentagem: currency): currency;
var
 valordesconto: currency;
begin
 valordesconto := 0;
 if Valor <= 10 then
 begin
  Promocao.Desconto := porcentagem;
  valordesconto := ((2 * Valor)/100);
  Valor := Valor - valordesconto;
 end;
 result := valordesconto;
end;

procedure TReajuste.ReajustarPromocaoLeiDemeter;
var
 produto: TProduto;
begin
 TotalDesconto := 0;
 for produto in Produtos do
 begin
  TotalDesconto := TotalDesconto + (produto.FazerReajuste(2));
 end;
end;
```

A primeira alteração realizada foi criar um método em TProduto que aplicasse o percentual de desconto e retornasse o quanto foi descontado. Assim, foi possível criar também o método ReajustarPromocaoLeiDemeter, que agora não viola seu princípio. Atente e veja que nesse método não estamos mais acessando o elemento TPromocao de TProduto, ele está encapsulado no método TProduto.FazerReajuste.

Hollywood Principle

Não nos chame, nós chamamos você. Este é conceito desse princípio. Mas, como isto está diretamente ligado na programação orientada a objetos? A ideia aqui é depender de

interfaces e não de objetos concretos, mas não só isso, se um método necessita de um determinado tipo de objeto em seu corpo, esse objeto não deve ser instanciado, mas sim passado como parâmetro. Vamos a outro exemplo simples, imagine uma classe de envio de e-mails (**Listagem 20**).

Listagem 20. Enviando emails

```
TEmail = class
end;

TConstrutorEmail = class
public
  function CriarEmail: TEmail;
end;

TNotificadorEmail = class
private
  FConstrutorEmail: TConstrutorEmail;
  procedure Enviar(email: TEmail);
public
  procedure EnviarEmail;
end;
```

Temos na **Listagem 20** uma classe TEmail que representa um e-mail que será construído pela classe TConstrutorEmail, digamos que esta verifica qual texto adicionar no email, formatações etc. Então temos TNotificadorEmail, que irá enviar o email no seu método EnviarEmail, veja-o a seguir:

```
procedure TNotificadorEmail.EnviarEmail;
var
 email : TEmail;
begin
 FConstrutorEmail := TConstrutorEmail.Create;
 email := FConstrutorEmail.CriarEmail;
 Enviar(email);
end;
```

Note que o próprio método instancia TConstrutorEmail, isso é sinal de acoplamento e é justamente isso também que o princípio Hollywood diz, não devemos instanciar uma classe dessa maneira em um método. A correção para isso é criar uma interface para TEmail e TConstrutorEmail e fazer com que nosso método TNotificadorEmail as utilize, como mostra a **Listagem 21**.

Como já falado, foram criadas as interfaces básicas e agora em TNotificadorEmailHollywood.EnviarEmail passamos como parâmetro uma instância de IConstrutorEmail, assim, o método está desobrigado de instanciar ele mesmo esse tipo. Isso é o princípio Hollywood. Existem Frameworks de IoC (Inversion of Control) que podem automatizar a instanciação de objetos de dependência, mas até o momento em que escrevo este artigo não conheço nenhum disponível para Delphi Win32.

KISS Principle

Keep It Simple, Stupid – Mantenha isso simples, idiota. O que esse princípio sugere é, codifique o código mais simples que resolva o problema. Código simples não significa código ruim, mas sim, simples mesmo. Um exemplo banal é que se você precisar percorrer uma lista, do primeiro ao quarto elemento, o que você usa? Um comando For correto? Sim, mas tem gente que vai utilizar um While para essa mesma tarefa. Porque complicar? E assim também é para as demais rotinas de seu sistema, seja prático e simples, não complique código (e seu entendimento) com desculpas de desempenho de processamento, a não ser claro, que o desempenho seja realmente visível a olho nu.

Listagem 21. Utilizando o princípio Hollywood

```
IEmail = interface
  ['{A1DF79B4-39AE-4591-83B8-F5CFBCA8F737}']
end;

IConstrutorEmail = interface
  ['{C761C74B-513B-4048-9B43-B6932209175A}']
  function CriarEmail: IEmail;
end;

TEmailHollywood = class(TInterfacedObject, IEmail)
end;

TConstrutorEmailHollywood = class(TInterfacedObject, IConstrutorEmail)
public
  function CriarEmail: IEmail;
end;

TNotificadorEmailHollywood = class
private
  procedure Enviar(email:IEmail);
public
  procedure EnviarEmail(construtor: IConstrutorEmail);
end;
```

YAGNI

You Ain't Gonna Need It – Você não vai precisar disso. Este princípio é bem simples. Não codifique o que você acha que não vai precisar, seja uma rotina, uma classe, uma possível situação. Faça o que você tem certeza no momento, e caso seja necessário, posteriormente você refatora seu código para incluir algo novo. Ou seja, não tente prever código. Se um usuário te pede X, entregue X e não XYZ.

DRY

Don't Repeat Yourself – Não se repita. Aqui tratamos o código duplicado. Duplicar uma lógica pelo sistema prejudica sua manutenção e estabilidade, porque caso você precise ajustar essa lógica que está duplicada, e esquecer algum ponto, seu sistema terá duas respostas diferentes para a mesma lógica.

SoC

Separation of Concern – Separação de responsabilidades. É uma maneira de se aplicar o Single Responsability Principle. Ele diz que temos que organizar nossas classes

Nota do DevMan

Frameworks de inversão de controle são frameworks que podem resolver a dependência de uma classe baseando-se na interface registrada dessa dependência. Esses frameworks são amplamente utilizados por desenvolvedores .NET e Java, pois esses ambientes oferecem diversos frameworks. Os mais conhecidos são Windsor, String, NInject e Unity. Sendo os três primeiros open source e o terceiro da Microsoft. Devido à inexistência disso no ambiente Delphi Win32, vou demonstrar o uso do Windsor em Delphi Prism. Imagine a seguinte classe:

```
type
 Reporter = public class
 private
  _reportBuilder: IReportBuilder;
  _emailSender: IEmailSender;
 public
  constructor (reportBuilder: IReportBuilder; emailSender: IEmailSender);
  method SendReport;
 end;

implementation

constructor Reporter(reportBuilder: IReportBuilder; emailSender: IEmailSender);
begin
 _reportBuilder := new ReportBuilder()
 _emailSender := new EmailSender()
end;

method Reporter.SendReport;
begin
 var email:= _reportBuilder.CreateReport();
 _emailSender.Send(email)
end;
```

A classe Reporter possui em seu construtor dependências a duas interfaces, IReport Builder e IEmailSender. O que um Framework IoC faz é que, ao instanciar o objeto Reporter, será detectado que ele precisa de um IReportBuilder e IEmailSender e eles serão instanciados automaticamente pelo Framework. Para saber quem implementa uma determinada interface é necessário fazer seu devido registro, como é mostrado a seguir:

```
var container := new WindsorContainer();
container.AddComponent("reportBuilder", typeof(IReportBuilder), typeof(ReportBuilder));
container.AddComponent("emailSender", typeof(IEmailSender), typeof(EmailSender));
container.AddComponent("reporter", typeof(Reporter));
```

Aqui estamos registrando quem implementa as interfaces IReportBuilder e IEmail-Sender. Assim o framework toma conhecimento de quem implementa o que, e podemos então pedir ao mesmo que resolva para nós o objeto Reporter, devolvendo uma instância pronta para ser utilizada:

```
var reporter := container.Resolve<Reporter>();
reporter.SendReport;
```

O devido uso desse tipo de framework favorece o baixo acoplamento de todo o sistema.

de forma coerente as suas responsabilidades dentro do sistema. E que essas responsabilidades não devem se misturar, formando assim camadas de um sistema. Teoricamente temos três camadas em um sistema: apresentação, regras de negócio e acesso a dados, nessa ordem mesmo. As camadas inferiores não devem acessar camadas superiores, ou seja, não tem porque um objeto da camada de acesso a dados acessar outro objeto que está lá na camada de apresentação.

Conclusão

Estes são os principais princípios, conceitos divulgados pela comunidade de desenvolvimento. Como podem ver, desenvolver um sistema é uma arte, que envolve harmonia sim, entre suas classes, para se obter algo de qualidade. Muitos programadores iniciantes, recém formados de cursos técnicos acreditam que já são bons porque fizeram um sistema X ou Y. Se esses princípios não são seguidos, imagino a qualidade de seus sistemas. Desenvolver um sistema não é só codificar, arrastar-clicar, é pensar no todo, é arquitetar peças, por isso, nunca pare de estudar, principalmente arquitetura de software. ●

Links

Castle Project
http://www.castleproject.org/

Spring.NET Framework
http://www.springframework.net/

NInject
http://ninject.org/

Unity
http://unity.codeplex.com/

Principles of OOD
http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Blogs do autor
http://pauloquicoli.spaces.live.com
http://twitter.com/pauloquicoli
http://www.nhibernatebrasil.net/

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Crie um agendador de tarefas OO – Parte 2

Refatorando o código

Rafael Stavarengo
faelsta@gmail.com
http://www.devmedia.com.br/stavarengo
Programador-analista da Cheina Informática. Trabalha com Delphi, JME e PHP + ZendFramework. Formando em Tecnologia de Análise e Desenvolvimento de Sistemas pela UNIPAR.

Resumo do DevMan

Esta segunda parte da série "Crie um agendador de tarefas OO" aborda conceitos diferentes da primeira parte. O artigo anterior apresentou os conceitos básicos da orientação a objetos sem se preocupar com a arquitetura que estava sendo criada - sem aplicar padrões de projetos. Já este segundo artigo começa a refatorar o código da primeira versão a fim de corrigir os problemas de engenharia existentes.

Na parte 1 deste artigo a primeira versão do agendador ficou pronta, mas com uma série de problemas. Não estou falando de erros, pois a aplicação funcionava sem gerar erro algum. Estou falando de problemas estruturais – alguns chamariam de problemas administrativos. Embora o agendador esteja cumprindo com sua obrigação de executar tarefas em horários pré-programados, estruturalmente ele está em péssimas condições. Não é necessário observar

Do que trata o artigo
Este artigo fala de técnicas de refatoração. Ensina como identificar e corrigir problemas de engenharia no software. É claro que para corrigir estes problemas alguns princípios de orientação a objetos também são utilizados. Para atingir o objetivo você também vai ver como criar métodos abstratos e como sobrescrever métodos. Além de entender para que serve e qual a diferença entre as instruções abstract, virtual, dynamic e override.

Para que serve
Refatorar é o processo de mudar a estrutura interna do software sem alterar seu comportamento externo. Esta técnica evita que o software se desgaste durante a vida útil do mesmo. Com esta técnica você consegue melhorar seu software sem que os clientes se sintam perdidos em cada alteração que você fizer.

Em que situação o tema é útil
Este tema é muito útil para sistemas com código desorganizados, pois a refatoração vai ajudar a melhorar o entendimento do código além de evitar a inclusão de novos defeitos durante futuras modificações.

muito para perceber os problemas de engenharia presentes no código. No próximo parágrafo você encontra uma breve descrição dos problemas que o código do sistema possui. Desta maneira, se você ainda não leu a primeira parte desta série poderá ler este artigo sem ficar perdido no assunto.

O maior problema da primeira versão é que toda a lógica do programa está dentro da classe TForm1, ou seja, dentro do Form1. Tudo está neste formulário: salvar tarefas, salvar agendamentos, monitorar os agendamentos, cálculo da próxima execução de uma tarefa, cálculo da próxima execução de um agendamento, cálculo para saber se está na hora de executar uma tarefa e até mesmo o método para executar as tarefas.

"Bad smells in code"

Nota

Se você tiver o código fonte da primeira versão, esse vai ajudar a entender os próximos parágrafos. Junto com os arquivos disponíveis para download está o código fonte da primeira versão do agendador. Caso você não tenha, poderá baixá-los.

Quando eu disse que tudo estava dentro da classe TForm1, eu não estava exagerando. Repare nos itens que eu citei no último parágrafo da introdução: o que mais a primeira versão faz além do que foi citado lá? Não faz mais nada. Tudo que o agendador faz foi escrito dentro da classe TForm1. Embora existam mais duas classes além da TForm1 – TTarefa e TAgendador – nenhuma vantagem da orientação a objetos foi usada. A classe TForm1 está muito grande e cheia de desvios feitos através de testes (instruções IF). Qualquer alteração no programa resultará em uma modificação na classe TForm1 e como esta classe está cheia de responsabilidades interligadas, qualquer modificação feita nela significa que outra parte da aplicação pode começar a gerar erros por conta disto – por isto todas as funções da aplicação devem ser testadas novamente.

Estes são um dos motivos para criar um sistema cujas responsabilidades estão separadas em classes, porém, existem muitos outros. Imagino eu que, se você já leu até aqui é porque você conhece os problemas de uma codificação ruim e deve estar mais interessado em como fazer da maneira certa do que nas consequências de se fazer da maneira mais fácil. Por isto vou direto ao ponto. Veja o que é e como refatorar um código, usando como exemplo a classe TForm1 do agendador.

Vantagens das classes com responsabilidade única

A classe TForm1 está cheia de responsabilidades e isto não é bom. Isto é tão ruim que até existe um princípio da orientação a objetos que trata especificamente disto, chama-se: Princípio da Responsabilidade Única. Para seguir este princípio é preciso identificar todas as responsabilidades de uma classe e separar estes objetivos em mais classes a qualquer custo. Sim, é mais fácil falar do que fazer. O princípio diz o que deve ser feito, porém, como é que se faz para identificar e separar as responsabilidades de uma classe? Antes de aprender como faz, veja as vantagens que isto vai trazer.

A primeira vantagem é que quando qualquer classe possui apenas uma responsabilidade, isto significa que ela também possui apenas uma única razão para mudar. Quando as classes do seu sistema possuem apenas uma única responsabilidade você consegue fazer alterações, adicionar ou remover novos comportamentos no sistema sem ficar preocupado se o resto do sistema vai ser influenciado, pois cada classe do sistema tem sua responsabilidade e sabe como realizá-la indiferentemente do que acontece com as demais classes. Em outras palavras é possível alterar o sistema sem precisar testar o sistema inteiro novamente: você vai precisar testar somente a parte que foi alterada ou adicionada.

Como saber se código está "cheirando mal"

Cheirar mal é um termo comumente usado para referenciar algo de errado em algum código. Para conseguir identificar as responsabilidades extras de uma classe, primeiro você deve saber qual o objetivo dela, ou seja, você precisa conhecer o sistema ao qual ela pertence e qual o objetivo dele. No caso do agendador, a classe TForm1 tem o objetivo de controlar as ações do formulário Form1. Apenas isto. Então olhando para a imagem do Form1 e para o código da Unit1.pas eu vou procurar por todas as ações que o Form1 possui e separar em novas classes o código que não é necessário para o Form1 funcionar. A imagem do Form1 está disponível na **Figura 1**.

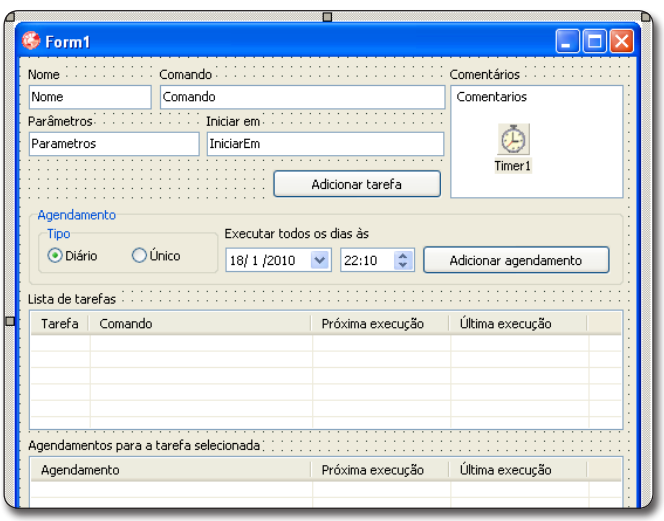


Figura 1. Imagem do Form1 que é controlado pela classe TForm1

Nota do DevMan

Refatoração é o processo de modificar o sistema para melhorar a estrutura do código sem alterar seu comportamento externo. Sempre que um pedaço de código da sua aplicação estiver "cheirando mal", você deve aplicar uma das quase 100 técnicas de refatoração disponíveis no site Refactoring Home Page. Este site é mantido pelo próprio Martin Fowler que é o autor do livro Refactoring: Improving the Design of Existing Code - ISBN 0201485672 - um dos livros mais importantes sobre o assunto. O link do site está no final do artigo.

Tenha em mente o que estou fazendo. Primeiro eu cheguei à conclusão de que a primeira versão do agendador, embora esteja funcionando bem, está muito rígida e por isto enfrentarei muitas dificuldades para mudar alguma coisa no programa durante a vida dele. Eu sei que terei dificuldades para fazer alterações no programa porque nenhum princípio da orientação a objetos foi seguido e toda a lógica da aplicação está em um único lugar – na classe TForm1. Então para tentar amenizar este problema é necessário implementar alguns princípios. Para começar eu escolhi implementar o princípio que a meu ver é base para qualquer outro princípio: o princípio da responsabilidade única.

O sistema possui três classes: TAgendador, TTarefa e TForm1. As duas primeiras classes citadas estão funcionando bem, e nenhum princípio precisa ou tem como ser aplicado a elas, pois elas são extremamente simples (por enquanto). Por outro lado, não é possível dizer o mesmo da classe TForm1. Ela está cheia de responsabilidades além da sua responsabilidade original.

Aprenda a identificar a reponsabilidade de uma classe

Sabendo qual é a verdadeira responsabilidade da classe TForm1, fica fácil identificar quais são as responsabilidades extras dela. Basta fazer uma lista do que a classe TForm1 precisa para fazer o Form1 funcionar, o que estiver na classe, mas não estiver na lista, é responsabilidade extra. Como eu disse, para identificar quais são as ações do Form1 que a classe TForm1 precisa executar, vou re-avaliar o código da classe para tentar identificar quais os trechos de código que podem ser separados. Algumas das listagens que vou mostrar a seguir também existem na primeira parte deste artigo (na edição anterior desta revista). Como as listagens já foram explicadas detalhadamente na edição anterior, não vou explicá-los novamente, mas para ajudar eu vou citar em qual listagem da última edição você vai encontrá-la. Se você precisar de mais explicações, basta procurar pela listagem citada na primeira parte do artigo. Comecemos pela **Listagem 1.**

O código da **Listagem 1** é uma das funções extras que a classe TForm1 está executando. Neste código está o teste necessário para saber quando será a data da próxima execução de um agendamento. Repare que para cada tipo de agendamento é feito um cálculo diferente, já que eles possuem informações diferentes quanto ao momento em que foram programados para a execução. Por exemplo, o agendamento diário armazena apenas a hora em que deve ser executado todos os dias e o agendamento único por sua vez armazena o dia e a hora do dia em que ele será executado.

É um fato que cada agendamento armazena informações diferentes e por isto é necessário fazer cálculos diferentes para descobrir quando é data da próxima execução. Atualmente a classe TForm1 é a única que sabe como fazer este cálculo. Se mais tipos de agendamentos forem adicionados ao agendador, então o código da **Listagem 1** com certeza ficará maior e mais difícil de entender. Este é o primeiro pedaço de código que eu identifiquei como responsabilidade extra da classe TForm1.

Uma observação importante

Raciocine comigo. Embora o agendador possua apenas uma classe para agendamentos chamada TAgendamento,

Listagem 1. Gerando a data da próxima execução do agendamento

```
01  function TForm1.GetDataDaProximaExecucao(Ag: TAgendamento): TDateTime;
02  Var
03    HoraDaExecucao, HoraAtualSemSegundos: TTime;
04    HoraDaÚltimaExecucao: TTime;
05    DataDaExecucao, DataAtualSemSegundos: TTime;
06    DataDaÚltimaExecucao: TTime;
07  Begin
08    Result := 0;
09    case Ag.Tipo of
10      agDiario:
11      begin
12        HoraDaExecucao := EncodeTime(
13          HourOf(Ag.DataDaExecucao), MinuteOf(Ag.DataDaExecucao), 0, 0);

14        HoraAtualSemSegundos := EncodeTime(
15          HourOf(Time), MinuteOf(Time), 0, 0);

16        HoraDaÚltimaExecucao := EncodeTime(
17          HourOf(Ag.ÚltimaExecucao), MinuteOf(Ag.ÚltimaExecucao), 0, 0);

18        if (HoraDaExecucao >= HoraAtualSemSegundos) and
19          (HoraDaÚltimaExecucao < HoraDaExecucao) then
20        begin
21          Result := EncodeDateTime(
22            YearOf(Today), MonthOf(Today), DayOf(Today),
23            HourOf(Ag.DataDaExecucao), MinuteOf(Ag.DataDaExecucao), 0, 0);
24        end
25        else
26        begin
27          Result := EncodeDateTime(
28            YearOf(Tomorrow), MonthOf(Tomorrow), DayOf(Tomorrow),
29            HourOf(Ag.DataDaExecucao), MinuteOf(Ag.DataDaExecucao), 0, 0);
30        end;
31      end;
32    agUnico:
33    begin
34      DataDaExecucao := EncodeDateTime(
35        YearOf(Ag.DataDaExecucao), MonthOf(Ag.DataDaExecucao),
36        DayOf(Ag.DataDaExecucao), HourOf(Ag.DataDaExecucao),
37        MinuteOf(Ag.DataDaExecucao), 0, 0);

38      DataAtualSemSegundos := EncodeDateTime(
39        YearOf(Now), MonthOf(Now), DayOf(Now), HourOf(Now),
40        MinuteOf(Now), 0, 0);

41      DataDaÚltimaExecucao := EncodeDateTime(
42        YearOf(Ag.ÚltimaExecucao), MonthOf(Ag.ÚltimaExecucao),
43        DayOf(Ag.ÚltimaExecucao), HourOf(Ag.ÚltimaExecucao),
44        MinuteOf(Ag.ÚltimaExecucao), 0, 0);

45      if (DataDaÚltimaExecucao = 0) and
46        (DataDaExecucao >= DataAtualSemSegundos) then
47        Result := DataDaExecucao
48      else
49        Result := 0;
50    end;
51  end;
52  end;
```

aparentemente esta classe também está sobrecarregada de responsabilidade. A classe TAgendamento está sendo ao mesmo tempo, todos os tipos de agendamentos existentes no sistema. Com o tempo mais agendamentos surgirão e a classe TAgendamento pode não conseguir atender a todas as peculiaridades de cada tipo de agendamento que podem surgir. Não há como prever quais os tipos de agendamentos vão aparecer, mas é possível prever que vão existir outros tipos (e ficar preparado para isto). Por isto, talvez seja mais interessante que existam classes de agendamento especializadas em cada tipo de agendamento existente.

Separando as responsabilidades de uma classe

É necessário remover/separar da classe TForm1 o código responsável por calcular a data da próxima execução dos agendamentos, pois isto é responsabilidade dos agendamentos e não da classe que cuida da interação entre o formulário e o usuário. A classe TAgendamento deve ser capaz de saber gerenciar suas próprias informações e uma destas informações é a data da próxima execução. Não estou falando da data de agendamento, mas sim da data da próxima execução. Embora pareça ser a mesma coisa, existe uma grande diferença. Data do agendamento é a data em que o usuário programou o agendamento. Esta informação é diferente para cada tipo de agendamento. Para o agendamento diário a data de execução é apenas a hora do dia em que ele será executado. Por outro lado, a data da próxima execução é o momento em que o agendamento será executado. No caso do agendamento diário a data da próxima execução depende se a hora do dia em que ele foi programado já passou. Se passada a data, a próxima execução será "amanhã na hora programada", caso contrário, a data da execução será "hoje na hora programada".

Uma vez que a responsabilidade extra da classe TForm1 foi encontrada, falta apenas migrá-la para o seu lugar de destino. Neste caso o cálculo da próxima execução dos agendamentos será migrado para a classe TAgendamento, pois nada mais apropriado do que o agendamento saber calcular a sua própria data da próxima execução.

Antes de mostrar como vai ficar o código da classe TAgendamento, quero levantar uma questão. Estou fazendo toda esta mudança para seguir o princípio de responsabilidade única. Porém, se o código da **Listagem 1** simplesmente for mudado de classe, sem que mais nenhuma mudança seja feita, o princípio não estará sendo seguido. O problema de múltiplas responsabilidades não será resolvido. Ele apenas será mudado de lugar. O método GetDataDaProximaExecucao já possui mais de uma responsabilidade por si próprio, pois ele sabe calcular a data da próxima execução de mais de um tipo de agendamento. Então o correto é que existam várias classes de agendamentos. Uma para cada tipo de agendamento existente.

Se outra classe de agendamento for criada – é o que vai acontecer – o código da **Listagem 1** pode ser cortado ao meio, e cada classe de agendamento vai ficar com a responsabilidade de saber como é o seu próprio cálculo. Além disso, será muito mais fácil adicionar novos tipos de agendamentos ao programa. Da maneira como está, se novos tipos de agendamentos forem criados mais testes terão que ser espalhados por todo o sistema, onde acontecem coisas diferentes para cada tipo de agendamento. Com classes de agendamentos especializadas esse problema não existe, pois cada uma delas vai saber como deve executar suas próprias funções independentemente de como as outras partes do sistema funcionam.

Por isto não vou simplesmente mudar o código da **Listagem 1** para a classe TAgendamento. Ao invés disso eu vou criar um grupo de classes especializadas em agendamentos. Veja o design deste grupo na **Figura 2**.

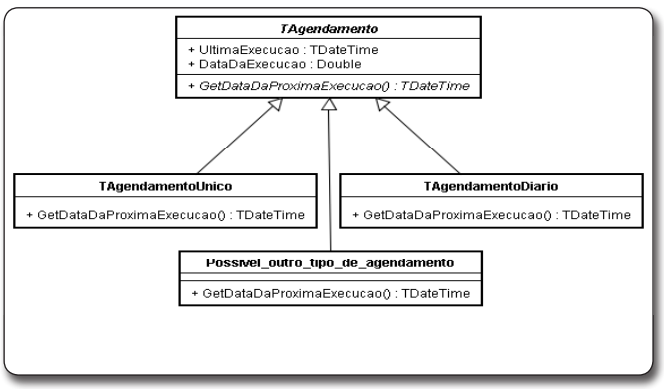


Figura 2. Novo design para o grupo de agendamentos

Grupo de classes para os agendamentos

Primeiro você precisa entender o que o diagrama de classes da **Figura 2** está mostrando. A classe TAgendamento foi abstraída, ou seja, agora ela não é mais uma classe que funciona sozinha. Ao contrário disto a classe TAgendamento agora é a super classe de um grupo de classes para os tipos de agendamentos do sistema. Observe que o nome da classe TAgendamento está em itálico. Isto significa que ela é abstrata. E por que ela é uma classe abstrata? Porque um dos seus métodos é abstrato. É o caso do método GetDataDaProximaExecucao. Veja que ele também está em itálico.

Quando um método é abstrato significa que ele não possui implementação, ou seja, ele não possui código. Ele possui apenas a assinatura e fica a cargo das subclasses criarem uma implementação para ele. Por isso as subclasses da classe TAgendamento redeclaram (sobrescrevem) o método GetDataDaProximaExecucao. São elas que definem o código dele. Outra coisa importante é que você jamais conseguirá criar uma instância de uma classe abstrata, ou seja, você não vai conseguir usar a instrução TAgendamento.Create. Quando fizer isso uma exceção será disparada dizendo que você não pode criar um objeto desta classe porque ela possui métodos abstratos.

Nota

O diagrama da **Figura 2** dá uma dica de como será simples implementar novos tipos de agendamentos ao sistema.

Codificando as novas classes de agendamentos

Agora que você entendeu o diagrama de classes da **Figura 2**, você está pronto para ver o código destas classes (**Listagem 2** e **Listagem 3**).

Listagem 2. Antiga classe TAgendamento

```
01 unit Agendamento;
02 interface
03 type
04  TTipoAgendamento = (agDiario, agUnico);
05  TAgendamento = class
06  public
07    UltimaExecucao: TDateTime;
08    DataDaExecucao: Double;
09    Tipo: TTipoAgendamento;
10  end;
11 implementation
12 end.
```

Listagem 3. A nova super classe TAgendamento

```
01 unit Agendamento;
02 interface
03 type
04  TAgendamento = class
05  public
06    UltimaExecucao: TDateTime;
07    DataDaExecucao: Double;
08    function GetDataDaProximaExecucao: TDateTime; virtual; abstract;
09  end;
10 implementation
11 end.
```

A **Listagem 2** mostra como era a classe TAgendamento até agora. Três coisas mudaram na classe TAgendamento para que ela possa se tornar super classe do grupo de classes de agendamento. Primeiro; como existirá uma classe especializada em cada tipo de agendamento existente, não é mais necessário usar o atributo "Tipo" que você pode ver na linha 9 da **Listagem 2**. Depois, por consequência disso o tipo enumerado TTipoAgendamento que está declarado na Linha 4 também foi removido. E por último o método abstrato GetDataDaProximaExecucao foi declarado, como você pode ver na linha 8 da **Listagem 3**. As palavras reservadas virtual e abstract são comuns em métodos abstratos. A instrução abstract diz ao compilador que o método não tem corpo, apenas a assinatura. A instrução virtual diz ao compilador que o método pode ser sobrescrito pelas subclasses, afinal do que adianta um método sem corpo se ele não puder ser sobrescrito em algum momento?

Repare que entre as linhas 10 e 11 da **Listagem 3** deveria estar a implementação do método GetDataDaProximaExecucao, porém esta implementação não existe, uma vez que métodos abstratos devem ser implementados pela subclasses da classe que fez a declaração. Se você tentar digitar algum código para este método dentro da classe TAgendamento, a programa não vai compilar.

As classes de agendamentos especializadas

A super classe do grupo de agendamentos já foi definida. Agora é necessário criar as subclasses para este grupo. Por enquanto vou criar apenas as classes TAgendamentoUnico e TAgendamentoDiario, como você pode ver no diagrama de classes da **Figura 2**. Além disso, o próprio diagrama já dá uma dica de como será simples criar mais tipos de agendamentos para o programa. Está na hora de criar as subclasses de TAgendamento (**Listagem 4**).

Listagem 4. Subclasse TAgendamentoUnico

```
01 unit AgendamentoUnico;
02 Interface
03 uses Agendamento, Tarefa;
04 Type
05  TAgendamentoUnico = class(TAgendamento)
06  Public
07    constructor Create(DataHora: TDateTime);
08    function GetDataDaProximaExecucao: TDateTime; override;
09  end;
10 Implementation
11 uses SysUtils, DateUtils;
12 { TAgendamentoUnico }
13 constructor TAgendamentoUnico.Create(DataHora: TDateTime);
14 Begin
15  inherited Create;
16  Self.DataDaExecucao := DataHora;
17 end;
18 function TAgendamentoUnico.GetDataDaProximaExecucao: TDateTime;
19 Var
20  HoraDaExecucao, HoraAtualSemSegundos: TTime;
21  HoraDaUltimaExecucao: TTime;
22  DataDaExecucao, DataAtualSemSegundos: TTime;
23  DataDaUltimaExecucao: TTime;
24 Begin
25  DataDaExecucao := EncodeDateTime(YearOf(Self.DataDaExecucao), MonthOf
26    (Self.DataDaExecucao), DayOf(Self.DataDaExecucao), HourOf(Self.DataDaExecucao),
27    MinuteOf(Self.DataDaExecucao), 0, 0);
28  DataAtualSemSegundos := EncodeDateTime(YearOf(Now), MonthOf(Now), DayOf(Now),
29  HourOf(Now), MinuteOf(Now), 0, 0);
30  DataDaUltimaExecucao := EncodeDateTime(YearOf(Self.UltimaExecucao),
31    MonthOf(Self.UltimaExecucao),DayOf(Self.UltimaExecucao), HourOf(Self.UltimaExecucao),
32    MinuteOf(Self.UltimaExecucao), 0, 0);

33  if (DataDaUltimaExecucao = 0) and (DataDaExecucao >= DataAtualSemSegundos)
34   Then
35   Result := DataDaExecucao
36  Else
37   Result := 0;
38 end;
39 end.
```

A **Listagem 4** mostra uma subclasse da classe TAgendamento. Eu posso dizer que TAgendamentoUnico é uma subclasse da classe TAgendamento por causa da instrução que aparece na linha 5. Observe a declaração dizendo que TAgendamentoUnico é igual a uma classe de TAgendamento. Em Delphi Language é assim que se cria herança entre classes. Herança é o que permite a uma classe herdar as características de outra. É como na herança genética, onde o filho herda características do pai e da mãe. Quando dizemos que TAgendamentoUnico é filha ou subclasse de TAgendamento, significa que TAgendamentoUnico herdou todas as características que TAgendamento possui. Como você viu na **Listagem 3**, a classe TAgendamento possui três características que serão herdadas por todas as suas filhas, inclusive TAgendamentoUnico. São elas: os dois atributos UltimaExecucao e DataDaExecucao, além do método abstrato GetDataDaProximaExecucao.

Se o método GetDataDaProximaExecucao fosse um método comum, ou seja, se não estivesse declarado como abstrato na superclasse, não seria necessário redeclará-lo nas classes filhas. Porém este não é o caso. A superclasse declarou este método como sendo abstrato e isso obriga todas as classes filhas a criar uma implementação para este método. Se você não implementar os métodos abstratos uma exceção é gerada quando você tentar criar um objeto da classe. Todas as linguagens que conheço seguem esta regra. Esta regra é exatamente o que o agendador precisa. Desta maneira cada subclasse de TAgendamento é obrigada a implementar o cálculo da próxima data de execução ao invés de colocar todos os tipos de cálculos dentro da classe TAgendamento. Em outras palavras, cada subclasse de agendamento é obrigada a implementar seu próprio cálculo da data de próxima execução.

Veja entre as linhas 4 e 9 da **Listagem 4** que não foi necessário redeclarar os atributos que estão declarados nas linhas 6 e 7 da **Listagem 3**. Porém, mesmo sem redeclará-los eles são usados por quase toda a implementação do método GetDataDaProximaExecucao. Esta é a técnica chamada de herança sendo usada na prática, onde uma classe herda todas as características da classe pai – também chamada de super classe.

Na **Listagem 5** está o código da subclasse TAgendamentoDiario. Não farei nenhuma explicação extra sobre esta classe, pois a única diferença entre a classe TAgendamentoDiario e a classe TAgendamentoUnico é o cálculo da próxima data de execução, ou seja, é a implementação do método GetDataDaProximaExecucao. A explicação sobre estes cálculos já foi feita na primeira parte desta série.

TForm1 não compila mais

O design do sistema mudou, pois eu criei mais duas classes para representar os tipos de agendamentos existentes. Com isto a classe TForm1 não compila e precisa de algumas mudanças para voltar a funcionar. Estas mudanças deixaram a classe mais simples. Veja na **Listagem 6** como fica o método AddAgendamentoClick da classe TForm1, antes e depois da refatoração.

Listagem 5. Subclasse TAgendamentoDiario

```
01 unit AgendamentoDiario;
02 interface
03 uses Agendamento, Tarefa;
04 type
05  TAgendamentoDiario = class(TAgendamento)
06  public
07   constructor Create(HoraDaExecucao: TTime);
08   function GetDataDaProximaExecucao: TDateTime; override;
09  end;

10 implementation
11 uses SysUtils, DateUtils;
12 { TAgendamentoDiario }
13 constructor TAgendamentoDiario.Create(HoraDaExecucao: TTime);
14 begin
15  inherited Create;
16  Self.DataDaExecucao := HoraDaExecucao;
17 end;
18 function TAgendamentoDiario.GetDataDaProximaExecucao: TDateTime;
19 var
20  HoraDaExecucao, HoraAtualSemSegundos: TTime;
21  HoraDaUltimaExecucao: TTime;
22 begin
23  HoraDaExecucao :=
24   EncodeTime(HourOf(DataDaExecucao), MinuteOf(DataDaExecucao), 0, 0);
25  HoraAtualSemSegundos :=
26   EncodeTime(HourOf(Time), MinuteOf(Time), 0, 0);
27  HoraDaUltimaExecucao :=
28   EncodeTime(HourOf(UltimaExecucao), MinuteOf(UltimaExecucao), 0, 0);
29  if (HoraDaExecucao >= HoraAtualSemSegundos)
30  and (HoraDaUltimaExecucao < HoraDaExecucao) then
31  begin
32  Result := EncodeDateTime(YearOf(Today), MonthOf(Today),
33    DayOf(Today), HourOf(DataDaExecucao), MinuteOf(DataDaExecucao), 0, 0);
34  end
35  else
36  begin
37   Result := EncodeDateTime(YearOf(Tomorrow), MonthOf(Tomorrow),
38    DayOf(Tomorrow), HourOf(DataDaExecucao), MinuteOf(DataDaExecucao), 0, 0);
39  end;
40 end;
41 end.
```

Listagem 6. Antes de depois da refatoração

```
01 //antes da refatoração
02 procedure TForm1.AddAgendamentoClick(Sender: TObject);
03 var
04  Ag: TAgendamento;
05 begin
06  Ag := TAgendamento.Create;
07  case TipoAgendamento.ItemIndex of
08   AGENDAMENTO_DIARIO:
09   begin
10    Ag.Tipo := agDiario;
11    Ag.DataDaExecucao := Hora.Time;
12   end;
13   AGENDAMENTO_UNICO:
14   begin
15    Ag.Tipo := agUnico;
16    Data.Time := Hora.Time;
17    Ag.DataDaExecucao := Data.DateTime;
18   end;
19  end;
20  GetTarefaSelecinada.Agendamentos.Add(Ag);
21  MostrarTarefas;
22 end;

23 //depois da refatoração
24 procedure TForm1.AddAgendamentoClick(Sender: TObject);
25 var
26  Ag: TAgendamento;
27 begin
28  Ag := nil;
29  case TipoAgendamento.ItemIndex of
30   AGENDAMENTO_DIARIO:
31   begin
32    Ag := TAgendamentoDiario.Create(Hora.Time);
33   end;
34   AGENDAMENTO_UNICO:
35   begin
36    Data.Time := Hora.Time;
37    Ag := TAgendamentoUnico.Create(Data.DateTime);
38   end;
39  end;
40  GetTarefaSelecinada.Agendamentos.Add(Ag);
41  MostrarTarefas;
42 end;
```

A diferença é sutil, mas as consequências são grandes. Como já disse, existem diversas classes de agendamentos, cada uma delas especializada em um tipo específico de agendamento. Com esta alteração, a classe TForm1 não precisa mais conhecer os detalhes das classes de agendamentos para poder criá-las. Ao invés disso a classe TForm1 apenas testa qual o tipo de agendamento o usuário escolheu e em seguida cria um objeto da classe de agendamento correspondente ao que o usuário selecionou, como você pode ver nas linhas 32 e 37 da **Listagem 6**.

Você percebeu que para criar um objeto de agendamento é necessário informar a hora da execução? Veja novamente a linha 7 das **Listagens 4** e **5**. Nelas está o código das classes TAgendamentoDiario e TAgendamentoUnico. Na linha 7 destas duas listagens está a declaração do método construtor das classes. Ambas as classes destas listagens possuem um parâmetro no método construtor. O construtor da classe TAgendamentoDiario possui um parâmetro chamado HoraDaExecucao do tipo TTime e o construtor da classe TAgendamentoUnico possui um parâmetro chamado DataHora do tipo TDateTime.

Criar os construtores com estes parâmetros não é obrigatório, mas por outro lado, facilita muito a utilização dos objetos destas classes. Pense: para que serve um agendamento se não for definido um momento de execução para ele? Não serve para nada. Um agendamento sem hora para execução, não chega nem a ser um agendamento, você concorda? De um jeito ou de outro o programador que criar objetos das classes de agendamentos deverá informar a hora de execução do mesmo. Ao colocar parâmetros no construtor da classe referente às informações obrigatórias que a classe deve possuir eu estou garantindo três coisas:

- Primeiro: será impossível para o programador esquecer-se de informar o valor dos atributos obrigatórios da classe, pois para criar um objeto da classe o compilador vai exigir o valor destes atributos;
- Segundo: será mais fácil trabalhar com os objetos da classe já que, o próprio método construtor já diz quais são os atributos obrigatórios para a classe;
- Terceiro: criar o objeto e definir os valores dos atributos obrigatórios será mais ágil, pois eu uso apenas uma linha de código para isto.

Nota

Ao colocar atributos no construtor das subclasses de agendamentos eu estou aplicando uma técnica de refatoração chamada Add Parameter. Lembre-se, existem mais de 100 técnicas de refatoração que você pode ver no site Refactoring Home Page (veja sessão Links).

Então recapitulando: definir o método construtor com parâmetros referente à hora de execução nas classes de agendamentos não é obrigatório, porém, eu só tenho a ganhar ao fazer isto, pois de nada adianta um objeto de agendamento sem ter a hora da execução definida. Veja na linha 16 das **Listagens 4** e **5** que o valor do parâmetro do construtor é passado ao atributo DataDaExecucao declarado na superclasse TAgendamento (**Listagem 3**).

Removendo métodos desnecessários de TForm1

A classe TForm1 ainda possui erros devido às alterações feitas no design do sistema. O método GetDataDaProximaExecucao (**Listagem 1**), por exemplo, está fazendo referência ao atributo Tipo da classe TAgendamento (veja linha 9 da **Listagem 1**). Como você sabe este atributo não existe mais e por isto é necessário mudar o código deste método. Na verdade, este método não é mais necessário dentro da classe TForm1. GetDataDaProximaExecucao era o método responsável por calcular a data da próxima execução de acordo com o tipo de agendamento, porém com o novo design a responsabilidade deste cálculo foi transferida para as subclasses de agendamentos. Agora cada classe de agendamento sabe calcular sua própria data de execução, uma vez que elas são especializadas nisto, ou seja, elas sabem fazer apenas isto: gerenciar as informações do tipo de agendamento que elas são especializadas. Depois de remover o método GetDataDaProximaExecucao eu vou substituir todas as chamadas do método removido por chamadas ao método das novas subclasses de agendamento.

Na **Listagem 7** está a implementação do método EstaNaHoraDeExecutar criado na versão 1 do agendador, ou seja, é como estava antes da refatoração – antes do novo design. Repare bem na linha 8. Ela está fazendo uma chamada ao método GetDataDaProximaExecucao, porém este não existe mais. Você provavelmente já sabe o que deve ser feito para solucionar este problema. O código da linha 8 deve ser substituído por:

```
ProximaExecucao := Ag.GetDataDaProximaExecucao;
```

Listagem 7. O antigo método TForm1.EstaNaHoraDeExecutar

```
01 function TForm1.EstaNaHoraDeExecutar(Ag: TAgendamento): Boolean;
02 var
03   Hora, Minuto, HoraAtual, MinutoAtual, Aux: Word;
04   ProximaExecucao: TDateTime;
05 begin
06   Result := False;
07   DecodeTime(Now, HoraAtual, MinutoAtual, Aux, Aux);
08   ProximaExecucao := GetDataDaProximaExecucao(Ag);
09   DecodeTime(ProximaExecucao, Hora, Minuto, Aux, Aux);
10   Result := (IsToday(ProximaExecucao) and (Hora = HoraAtual) and
11     (Minuto = MinutoAtual));
12 end;
```

Assim como a linha 8 existem outros lugares da classe TForm1 que fazem chamadas ao método GetDataDaProximaExecucao, que não existe mais. É claro que em todos estes lugares a chamada deve ser alterada para o método das subclasses de TAgendamento.

Convertendo um objeto em string

O método exibido na **Listagem 8** tem um objetivo muito simples. Ele exibe em uma lista todos os agendamentos da tarefa que foi selecionada pelo usuário. Em outras palavras, este método está fazendo com que as informações de um objeto do tipo TAgendamento sejam mais simples de entender para o ser humano. Porém existem dois problemas neste método.

Listagem 8. Refatorando o método TForm1.MostrarAgendamentos

```
01 procedure TForm1.MostrarAgendamentos;
02 var
03   Ag: TAgendamento;
04   Aux: TDateTime;
05 begin
06   ListaDeAgendamentos.Clear;
07   if GetTarefaSelecinada <> nil then
08   begin
09     for Ag in GetTarefaSelecinada.Agendamentos do
10     begin
11       with ListaDeAgendamentos.Items.Add do
12       begin
13         case Ag.Tipo of
14           agDiario:
15             Caption := 'Todos os dias às ' + FormatDateTime
16               ('HH:mm', Ag.DataDaExecucao);
17           agUnico:
18             Caption := 'Somente uma vez em ' + FormatDateTime
19               ('dd-mm-yy "às" HH:mm', Ag.DataDaExecucao);
20         end;
21         Aux := GetDataDaProximaExecucao(Ag);
22         SubItems.Add(IfThen(Aux = 0 , '<nunca>', DateTimeToStr(Aux)));

23         Aux := Ag.UltimaExecucao;
24         SubItems.Add(IfThen(Aux = 0 , '<nunca>', DateTimeToStr(Aux)));
25       end;
26     end;
27   end;
28 end;
```

O primeiro é que ele está fazendo uma chamada ao antigo método GetDataDaProximaExecucao, como você pode ver na linha 21. Para resolver este problema, basta substituir a chamada para o método da classe TAgendamento assim como eu expliquei anteriormente. O segundo problema está no código entre as linhas 13 e 20. O fato é que o método MostrarAgendamento está usando o atributo Tipo da classe TAgendamento e como você já sabe este atributo não existe mais. Ele usa este atributo para testar qual é o tipo de agendamento e então criar uma string amigável para o usuário.

Como você pode ver neste mesmo trecho de código, cada tipo de agendamento é representado de maneira diferente para o usuário, ou seja, cada tipo de agendamento possui uma string amigável diferente. Neste caso nada mais justo que cada subclasse de TAgendamento saiba criar sua própria string amigável que melhor represente suas informações. Desta maneira é possível remover mais este pedaço de código que não é de responsabilidade da classe TForm1.

O mais interessante é que não é necessário criar nenhum tipo de método extra nas subclasses de TAgendamento para este objetivo, pois nas versões mais recentes do Delphi a classe TObject possui um método virtual chamado ToString (lembre-se que virtual significa que ele pode ser sobrescrito). O objetivo deste método é retornar uma "string amigável" que represente um objeto qualquer. O fato de este método ser declarado como virtual na classe TObject significa que você pode sobrescrevê-lo. A implementação padrão deste método apenas retorna o nome da classe, porém a string TAgendamentoDiario não é o que se pode chamar de string amigável. A intenção é que cada classe de agendamento sobrescreva o método ToString da classe TObject para que ao ser invocado, este método retorne uma string que possa representar o agendamento em questão (**Listagem 9**).

A **Listagem 9** mostra a nova implementação do método ToString para cada subclasse de TAgendamento. Entre as linhas 1 e 4 está o método ToString da classe TAgendamentoDiario e nas linhas restantes está a implementação do método ToString para a classe TAgendamentoUnico. Uma vez que cada classe de agendamento sabe criar sua própria string amigável, o trecho de código que está entre as linhas 13 e 20 da **Listagem 8** pode ser substituído por apenas uma linha, como você pode ver na linha 13 da **Listagem 10**.

Listagem 9. Implementação do método ToString

```
01 function TAgendamentoDiario.ToString: string;
02 begin
03   Result := 'Todos os dias às ' + FormatDateTime('HH:mm', DataDaExecucao);
04 end;
05 function TAgendamentoUnico.ToString: string;
06 begin
07   Result := 'Somente uma vez em ' +
08     FormatDateTime('dd-mm-yy "às" HH:mm', DataDaExecucao);
09 end;
```

Listagem 10. Convertendo o objeto de agendamento para string

```
01 procedure TForm1.MostrarAgendamentos;
02   var
03   Ag: TAgendamento;
04   Aux: TDateTime;
05 begin
06   ListaDeAgendamentos.Clear;
07   if GetTarefaSelecinada <> nil then
08   begin
09     for Ag in GetTarefaSelecinada.Agendamentos do
10     begin
11       with ListaDeAgendamentos.Items.Add do
12       begin
13         Caption := Ag.ToString;
14         Aux := Ag.GetDataDaProximaExecucao;
15         SubItems.Add(IfThen(Aux = 0 , '<nunca>', DateTimeToStr(Aux)));
16         Aux := Ag.UltimaExecucao;
17         SubItems.Add(IfThen(Aux = 0 , '<nunca>', DateTimeToStr(Aux)));
18       end;
19     end;
20   end;
21 end;
```

Nota

A assinatura do método ToString mostrado na **Listagem 9** deve ser feita na sessão public da classe e obrigatoriamente deve estar acompanhado da palavra chave override.

O mais interessante disso tudo é que a classe TForm1 usa as classes especializadas de agendamento como se fosse uma classe comum de agendamento, ou seja, é possível usar as classes especializadas como se elas fossem uma única classe. Você percebe isto porque a classe TForm1 sempre declara a variável de agendamento como sendo do tipo TAgendamento. Veja linha 3 da **Listagem 10**. Assim a classe TForm1 consegue usar diversos tipos de agendamento diferentes sem precisar mudar o código, como era feito antes da refatoração.

Antes da refatoração, a classe TForm1 precisava fazer testes com o atributo Tipo da classe TAgendamento para que pedaços de códigos diferentes fossem escritos de acordo com o tipo de agendamento que estava trabalhando.

Para concluir as alterações da classe TForm1 afim de que ela volte a funcionar, falta remover a chamada do antigo método GetDataDaProximaExecucao que está dentro do método MostrarTarefas. Eu não vou mostrar o código deste método neste artigo por ser simples e intuitivo, mas você pode encontrá-lo junto com o código fonte completo para download no endereço deste artigo.

TForm1 voltou a compilar mas ainda está rígida

A classe TForm1 ainda está rígida. Sim, ela já não está mais tão rígida como antes, porém ainda está. Agora com o novo grupo de classes especializadas em agendamentos ficou muito mais simples trabalhar com agendamentos, pois as classes especializadas fazem quase todo o trabalho necessário.

Eu disse: quase todo o trabalho. Ainda existem muitas coisas que precisam ser separadas da classe TForm1. Até agora eu removi apenas duas responsabilidades extras da classe TForm1, são elas: o cálculo da próxima execução e a conversão do agendamento para uma string. Existem ainda mais responsabilidades extras na TForm1. Veja a seguir uma pequena lista de responsabilidades extras que devem ser removidas da classe TForm1:

• Salvar tarefas e agendamentos: Não é responsabilidade da classe TForm1 saber como salvar uma informação. Então de quem é? Vá pensando. Por enquanto vou dar uma dica. Também não é responsabilidade das classes de tarefa ou de agendamento. Você verá a reposta nas próximas edições da revista;

• Cálculo para saber se está na hora de executar uma tarefa: Veja o método EstaNaHoraDeExecutar apresentado na **Listagem 7**. A classe TForm1 executa uma série de instruções para poder testar se um determinado agendamento está na hora de ser executado. Você não vai querer este código espalhado por todo o seu sistema. Onde você vai colocá-lo? De quem deve ser esta responsabilidade?

• Monitorar os agendamentos e executar as tarefas na hora certa: A classe TForm1 possui um timer que fica monitorando os agendamentos e quando alguns destes agendamentos chegam na hora marcada, a própria classe TForm1 se encarrega de executar a tarefa relacionada ao agendamento. Com certeza nem uma destas duas coisas é responsabilidade da classe TForm1. De quem você acha que é?

• Cálculo da próxima execução de uma tarefa: Assim como não é responsabilidade da classe TForm1 calcular a data da próxima execução de um agendamento, também não é de responsabilidade dela saber calcular a data da próxima execução de uma tarefa. O cálculo da próxima execução de uma tarefa é bem simples. Basta olhar para todos os agendamentos da tarefa. Aquele que tiver a data mais recente será a próxima execução dela. Mesmo sendo simples, de quem é esta responsabilidade? Uma dica! Se o cálculo da próxima execução de um agendamento é responsabilidade das classes de agendamento, então de quem é a responsabilidade do cálculo da próxima execução de uma tarefa? (seria a classe TTarefa?)

Conclusão

O problema de múltiplas responsabilidades é algo que ocorre no cotidiano e às vezes nem nos damos conta. Conhecer as boas práticas da engenharia de sistemas nos ajuda a desenvolver software de qualidade. Neste artigo vimos um problema e sua solução. Essa solução levou a uma outra boa prática, a refatoração de código. Esta por sua vez, deve ser apoiada por testes automatizados, que irão garantir que não estaremos quebrando código ao ajustar alguma coisa. Mas, teste automático é tema para um outro artigo. Deixo a você leitor o desafio de se aprofundar. ●

Links

Refactoring Home mantido pelo próprio Martin Fowler
http://www.refactoring.com

Refactorings in Alphabetical Order
http://www.refactoring.com/catalog/index.html

Martin Fowler
http://pt.wikipedia.org/wiki/Martin_Fowler

Refatoração
http://pt.wikipedia.org/wiki/Refatoração

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Design Patterns para o mundo real

Template Method, Abstract Factory e Adapter

Do que trata o artigo
Este artigo trata da aplicação prática de design patterns em problemas reais que enfrentamos no dia-a-dia. Construiremos um pequeno framework orientado a objetos e ao longo do desenvolvimento iremos implementar alguns design patterns para resolver problemas reais do sistema.

Para que serve
A aplicação de design patterns pode aumentar a qualidade de seus sistemas, pois tratam-se de soluções testadas e aprovadas para problemas comuns do desenvolvimento de software. Com o uso de patterns você pode reduzir acoplamento, aumentar a coesão, melhorar a legibilidade de seu código e aumentar o grau de reuso do mesmo.

Em que situação o tema é útil
Design patterns são exatamente soluções para resolver problemas corriqueiros, ou seja, você pode usá-los a qualquer momento em seus projetos, seja na fase de desenvolvimento ou de manutenção você pode se deparar com um problema que já foi analisado e solucionado antes.

Ricardo Coelho
Possui graduação em T.I, desenvolve software em Delphi há cinco anos, estando envolvido principalmente em projetos ligados à telecomunicações. Hoje é colaborador da Sinapse Informática, onde faz parte da equipe de migração de um produto Win32 para Web utilizando C#. É também responsável pela definição, análise e melhoria dos processos de desenvolvimento da empresa.

Resumo do DevMan

Neste artigo iremos refatorar uma aplicação desenvolvida no artigo "Design Patterns para o mundo real, Singleton, Factory e DAO" (veja sessão Links), onde aplicaremos os patterns Template Method para criar classes base com um esqueleto de um algoritmo, Abstract Factory para criação de fábricas de famílias de objetos e também o Adapter para que possamos abstrair de nossa aplicação detalhes específicos de acesso a dados (nesse caso, dbExpress e ADO). Sugiro a leitura do artigo citado para um melhor entendimento e aproveitamento deste exemplo.

Os design patterns são padrões de design de soluções de software que já foram usados e aprovados para solucionar problemas comuns do desenvolvimento de software. Os patterns mais famosos são os que foram catalogados no famoso livro conhecido como GOF, apesar de esta literatura conter a descrição de 23 patterns, não significa que os design patterns se limitam a este grupo.

O uso destes patterns, tanto os descritos no GOF quanto outros, contribui para o aumento da qualidade do nosso software,

permitindo um menor grau de acoplamento, um maior reuso de código devido ao nível de abstração aplicado, uma maior coesão das classes e uma melhor legibilidade do código.

Os design patterns são independentes de plataforma e tecnologia, sendo possível aplicá-los em qualquer linguagem orientada a objetos. O conhecimento de orientação a objetos, por sinal, é algo imprescindível para a compreensão dos patterns. No artigo citado anteriormente, nós construímos uma aplicação usando os patterns Singleton, Factory e DAO. Neste artigo a dinâmica será a mesma. Primeiramente nós explicaremos os patterns que usaremos e depois falaremos sobre as alterações realizadas na aplicação anterior, com a implementação dos patterns pertinentes a cada tipo de problema.

Neste artigo usaremos o Template Method para permitir que determinadas partes de um algoritmo definido na classe base sejam implementados em classes filhas, usaremos o Abstract Factory para criamos fábricas de famílias de objetos, além de usarmos o Adapter para que possamos abstrair a camada de acesso a dados de forma que não seja traumática uma possível mudança no futuro.

Nota

No endereço para download deste artigo você pode encontrar o código-fonte da primeira versão do aplicativo, o que ajudará a compreender as adaptações, refatorações e aplicação dos patterns vistos neste artigo.

Template Method

O Template Method é um pattern comportamental que permite a definição do esqueleto de um algoritmo em uma classe base, porém com a definição de métodos abstratos que serão implementados nas classes filhas, permitindo que reutilizemos as partes em comum deste algoritmo e que reescrevamos apenas as partes diferentes para cada classe filha.

Para aplicarmos este pattern temos pelo menos duas classes envolvidas. A classe base (abstrata) e a classe filha (concreta). A primeira possui o método que define o esqueleto do algoritmo a ser executado e a segunda possui a implementação dos "pedaços" que faltam para a execução do método definido na classe pai.

Como podemos ver na **Figura 1,** temos a classe base com os métodos executaAcao, operacao1 e operacao2, onde executaAcao é o template method, é ele que define o algoritmo a ser executado. Na implementação do executaAcao teremos a chamada aos métodos acao1 e acao2 sendo que estes são abstratos e serão implementados nas classes concretas.

Abstract Factory

Abstract Factory é um pattern criacional que permite a criação de fábricas de famílias de objetos, que possuam alguma coisa em comum ou então uma relação entre si, sem especificarmos sua classe concreta. Entre seus benefícios podemos citar: isola classes concretas, facilita a troca de famílias de objetos e garante a consistência dos objetos que precisam sempre trabalhar em conjunto.

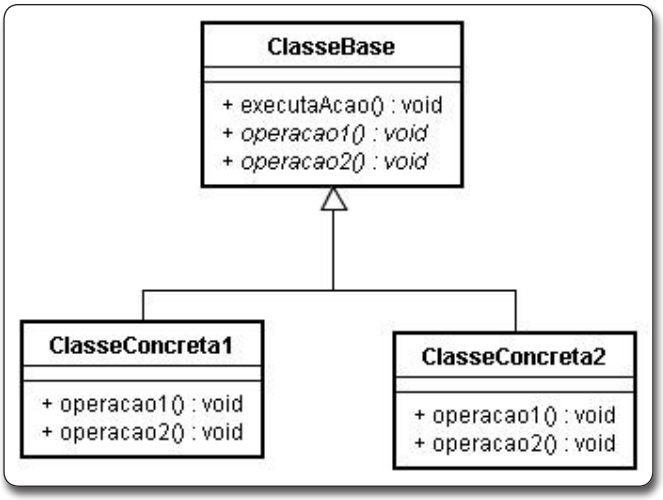


Figura 1. Template Method

Como podemos ver na **Figura 2**, temos uma AbstractFactory que é uma classe abstrata que possui o comportamento comum das nossas fábricas de objetos. As classes ConcreteFactory1 e ConcreteFactory2 são nossas fábricas que criam os objetos de fato. A primeira cria objetos da família 1 e a segunda cria objetos da família 2. Podemos observar também que o retorno de nossas fábricas são os AbstractProductA e AbstractProductB, sendo que cada classe retornará a respectiva implementação deste produto abstrato. Vemos também que o cliente não conhece nenhuma implementação de produto concreto.

Adapter

Este pattern faz com que nossa aplicação conheça uma interface única para acessar determinadas classes. Este pattern é interessante para evitar problemas como troca de versão de componentes de terceiros e também troca do componente em si por outro.

Observe na **Figura 3** onde temos a implementação do pattern Adapter. Temos a interface Target que é a classe que o cliente irá conhecer, Adapter é a classe que fará o redirecionamento ao Adaptee que é o objeto adaptado e que possui a implementação real do código. Caso haja alguma alteração nos métodos de Adaptee, só teremos que mexer no nosso Adapter, e caso queiramos trocar o Adaptee para usar outro framework, por exemplo, só precisaremos mexer no nosso Adapter, assim abstraímos mais a nossa aplicação e a tornamos mais fácil de ser alterada.

Remodelando o framework

No artigo "Design Patterns para o mundo real Singleton, Factory e DAO" nós construímos uma aplicação simples apenas com um cadastro de usuários, de clientes e uma tela de login e implementamos os patterns Singleton, DAO e Factory. Em nossa aplicação anterior tínhamos uma DAO para cada um de nossos modelos, estas DAOS implementam a interface IDAO e usam os serviços de conexão que, no nosso caso, estão encapsulados na classe TConexaoBD. Essa classe tem toda a configuração para conexão com o banco, além de uma instância de TSQLConnection que será fornecido às DAOs que solicitarem.

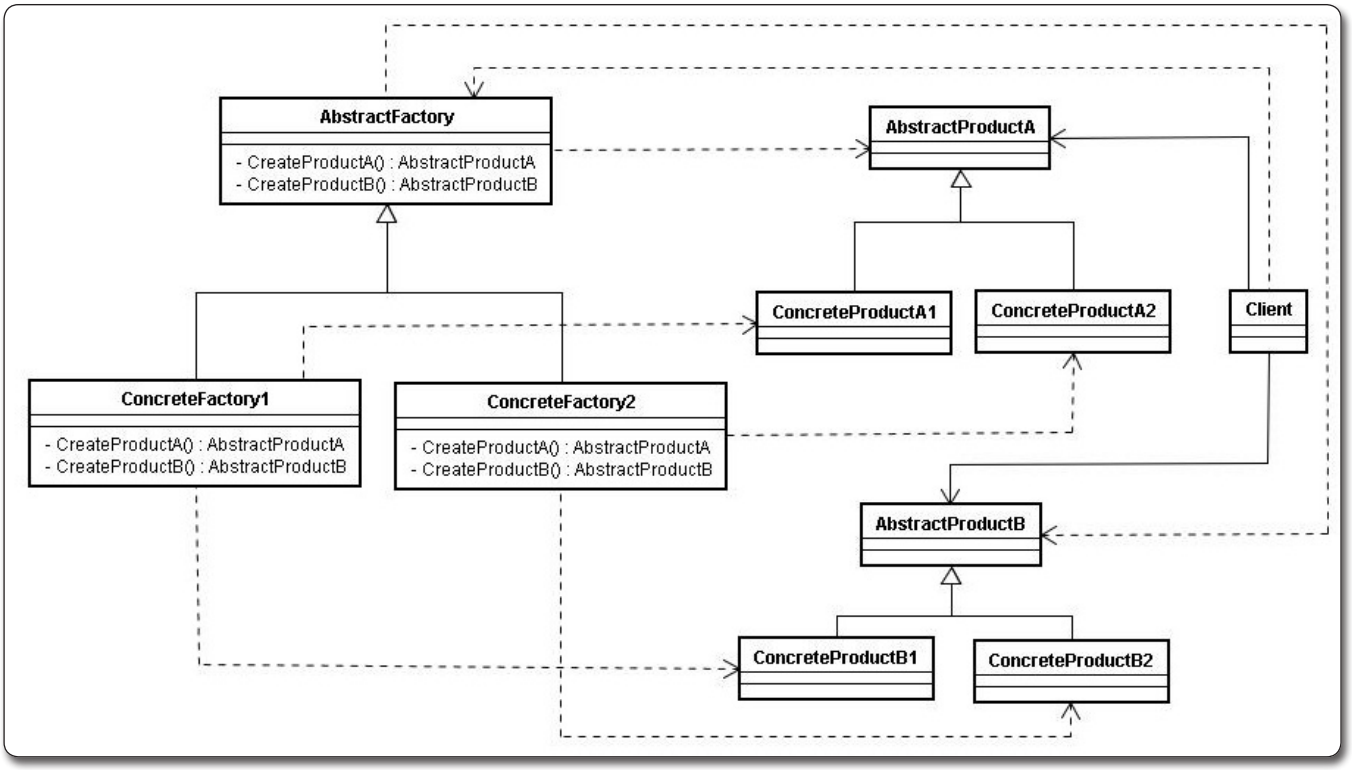


Figura 2. Abstract Factory

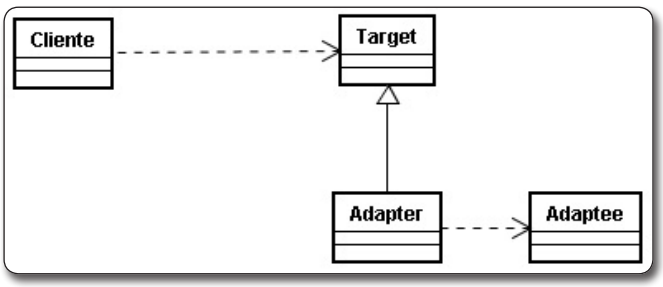


Figura 3. Adapter

Apenas com esta definição podemos ver que nossa aplicação está presa ao dbExpress. Vamos imaginar que o dbExpress fosse descontinuado, ou que surgisse no mercado algo muito melhor, ou que simplesmente fossemos usar outro engine que se comunica melhor com determinado banco. Teríamos que alterar no mínimo todas as DAOs.

No exemplo de hoje, alteraremos nosso framework para que possamos migrar e implementar facilmente suporte a outras tecnologias de conexão, como ADO, por exemplo, deixando nossa aplicação independente de uma tecnologia específica, para isso usaremos o pattern Adapter. Para que isso seja possível, criaremos também uma AbstractFactory para os objetos de banco de dados, pois nossas DAOs passarão a conhecer apenas nosso Adapter e a instanciação dos objetos de BD (Query's, Connections) ficará a cargo das fábricas concretas.

Na **Figura 4** podemos observar que agora o formulário base se comunica com a DaoFactory para obter um domínio a ser persistido pela tela. Além disso, temos também o DBObjectAbstractFactory, que é a nossa factory de fábricas concretas de objetos de banco de dados. Ela irá retornar uma fábrica de objetos de banco dados para uma tecnologia específica.

Na **Figura 5** podemos ver a estrutura de nossa AbstractFactory, onde temos IDBObjectFactory que determina o comportamento de nossas fábricas concretas, que são a DBXObjectFactory e AdoObjectFactory. Além disso, podemos ver também a estrutura do nosso Adapter. Observe que criamos uma classe abstrata TSqlQueryAdapter e duas classes concretas para esta, DBXAdapter e AdoAdapter. Nossa DAO agora conhece apenas a TSQLQueryAdapter e AdapterFactory, desconhecendo completamente a tecnologia de conexão que está sendo usada.

Classe de conexão

Nossa classe de conexão (**Listagem 1**) sofreu algumas alterações. Ela deixará de ter uma referência direta à TSQLConnection para ter uma referência à TCustomConnection (linha 03) - classe base da VCL de todos os componentes de conexão - além de ter seu método LoadConfig (linha 36) alterado para que os parâmetros de conexão sejam carregados de acordo com a tecnologia utilizada.

Além dessa mudança de referência direta à classe TSQLConnection, a forma de criação da conexão agora se dá através de nossa Factory (linha 26). O método LoadConfig esconde detalhes específicos referentes a configuração do dbExpress e ADO.

Note que temos as propriedades comuns a todos os tipos de conexão, como as definições de credenciais de acesso (usuário e senha – linhas 05 e 06). Dessa forma, nossa classe esconde detalhes inerentes a cada tipo de tecnologia. Além disso, o construtor é privado (linha 11, implementado nas linhas 22-29),

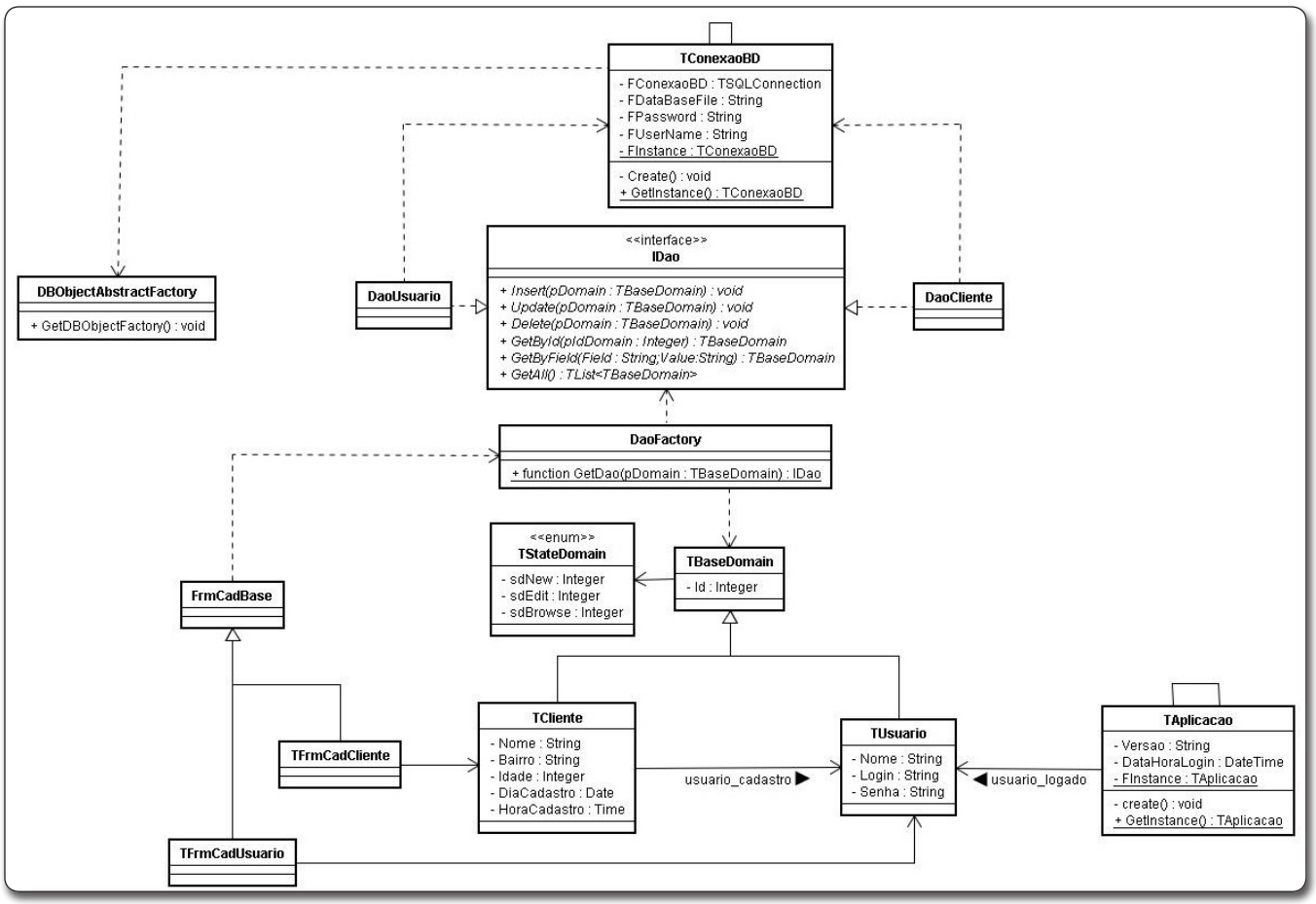


Figura 4. Diagrama da aplicação – Conexão, DAOs e formulários

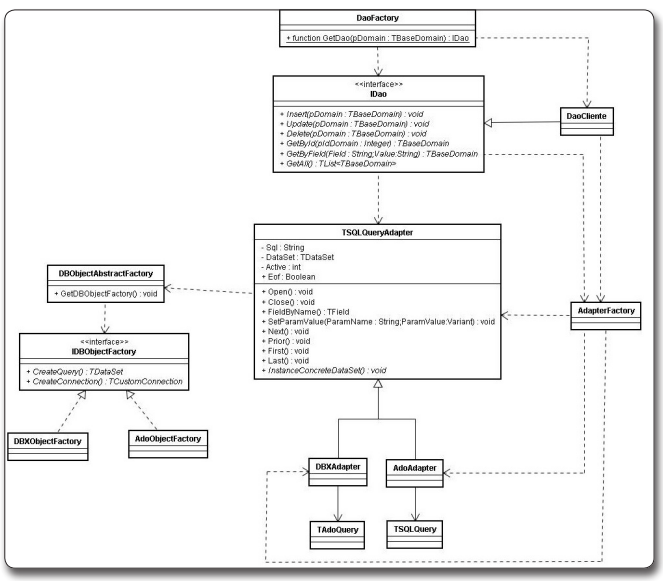


Figura 5. Diagrama da aplicação - Adapters e Abstract Factory

para que apenas a própria classe possa criar a real instância de conexão, internamente, retornado pelo método estático e público GetInstance (linhas 30-35).

Vale lembrar que a classe de conexão da **Listagem 1** foi criada no último artigo utilizando o padrão Singleton, portando os códigos das linhas 08, 11, 19 e 30-35 fazem parte deste padrão e servem para garantir que apenas uma instância deste objeto exista por vez.

A grande mudança está no método LoadConfig (linhas 36-73). Primeiro verificamos se a connection está criada para evitarmos um Access Violation (linha 39), em seguida verificamos o tipo de conexão usada na aplicação para configurarmos a mesma (linhas 42 e 61). No caso de conexão via DBX nós definimos o nome da conexão (linha 47), carregamos o nome do driver (linha 48) e utilizamos o método LoadParamsFromIniFile (linha 50) que carregará os parâmetros padrões de conexão com base no ConnectionName. Para finalizar, informamos os dados de usuário, senha e banco de dados (linhas 52-54), alteramos a propriedade LoginPrompt e abrimos a conexão (linhas 55 e 57).

Caso a conexão seja via ADO então apenas configuramos duas propriedades: ConnectionString que concatena o FILE NAME com a string de conexão passada através da propriedade DataBasefile (linha 66) e LoginPrompt (linha 67). Por fim abrimos a conexão (linha 69).

Factories

Na **Listagem 2** temos a nossa Factory de Factories. Esta factory é responsável simplesmente por verificar a tecnologia de acesso a dados usada e então criar a respectiva fábrica para

Listagem 1. Classe de conexão

```
...
TConexaoBD = class
 private
  FConexaoBD: TCustomConnection;
  FDataBaseFile: String;
  FPassword: String;
  FUserName: String;
  //Variável estática para armazenar a única instância de nossa classe
  class var FInstance:TConexaoBD;
  ...
  //Construtor privado para que possamos criar
  //o objeto apenas de dentro da própria classe
  constructor create();
 published
  property ConexaoBD:TCustomConnection read FConexaoBD write
  SetConexaoBD;
  property DataBaseFile:String read FDataBaseFile write SetDataBaseFile;
  property UserName:String read FUserName write SetUserName;
  property Password:String read FPassword write SetPassword;
 public
  //Método estático para retornar nosso objeto de conexão
  class function GetInstance():TConexaoBD;
  procedure LoadConfig();
 end;
...
constructor TConexaoBD.create;
begin
 inherited;
 //Criamos nossa connection que será fornecida para as DAOs
 ConexaoBD := TDBObjectAbstractFactory.GetDBObjectFactory().
  CreateConnection();
 //Carregamos as configurações de conexão
 LoadConfig;
end;
class function TConexaoBD.GetInstance: TConexaoBD;
begin
 if(not Assigned(FInstance))then
  FInstance := create;
 result := FInstance;
end;
procedure TConexaoBD.LoadConfig;
begin
 //Verificamos se o connection está criado para evitarmos um AccessViolation
 if(Assigned(ConexaoBD))then
 begin
 //verificamos o tipo de conexão usada na aplicação para configurarmos a mesma
  if(TAplicacao.TypeConnection = tcDbx)then
  begin
   if(not ConexaoBD.Connected)then
   begin
    //Definimos o nome da conexão para carregar as informações padrões de
      conexão
    TSQLConnection(ConexaoBD).ConnectionName := 'FBConnection';
    TSQLConnection(ConexaoBD).DriverName := 'Firebird';
    //LoadParamsFromIniFile carregará os parâmetros padrões de conexão com
      base no ConnectionName
    TSQLConnection(ConexaoBD).LoadParamsFromIniFile();
    //Definimos nossas configurações de usuário, senha e caminho do BD
    TSQLConnection(ConexaoBD).Params[TSQLConnection(ConexaoBD).
     Params.IndexOfName('Password')] := 'Password='+Password;
    TSQLConnection(ConexaoBD).Params[TSQLConnection(
     ConexaoBD).Params.IndexOfName('User_Name')] := 'User_Name='
     +UserName;
    TSQLConnection(ConexaoBD).Params[TSQLConnection(ConexaoBD).
     Params.IndexOfName('Database')] := 'Database='+DataBaseFile;
    ConexaoBD.LoginPrompt := false;
    //Abrimos a conexão
    ConexaoBD.Open;
   end;
  end
  else
  if(TAplicacao.TypeConnection = tcAdo)then
  begin
   if(not ConexaoBD.Connected)then
   begin
    //Definimos nossas configurações de usuário, senha e caminho do BD
    TADOConnection(ConexaoBD).ConnectionString := 'FILE NAME='+
    DataBaseFile;
    ConexaoBD.LoginPrompt := false;
    //Abrimos a conexão
    ConexaoBD.Open;
   end;
  end;
 end;
end;
```

objetos desta tecnologia, de acordo com o valor especificado no tipo enumerado TConnectionType, que pode ser tcDbx ou tcAdo. No nosso exemplo essa definição fica em uma constante pública, porém esta configuração poderia ficar em um arquivo INI ou XML, tornando a solução ainda mais dinâmica e flexível.

Na **Listagem 3** temos a interface para nossas factories, que é bastante simples. Aqui basicamente definimos o comportamento comum a todas as factories de objetos de banco de dados, como por exemplo, a capacidade de criar uma conexão e uma query (DataSet).

Nas **Listagem 4 e 5** temos respectivamente as fábricas concretas para dbExpress e ADO, onde cada uma cria e retorna seus objetos de banco de dados pertinentes a cada método. No

Dica

Uma melhoria que poderíamos implementar aqui é que, como as configurações para LoginPrompt e a chamada ao Open são comuns à todas as CustomConnections, elas poderiam ficar fora dos IFs e não precisariam ser replicadas para cada tipo.

Listagem 2. Implementação da Factory TDBObjectAbstractFactory

```
...
TDBObjectAbstractFactory = class
 public
  class function GetDBObjectFactory():IDBObjectFactory;
end;
...
class functionTDBObjectAbstractFactory.GetDBObjectFactory: IDBObjectFactory;
begin
 if(TAplicacao.TypeConnection = tcDbx)then
  Result := TDbxObjectFactory.create
 else
 if(TAplicacao.TypeConnection = tcAdo)then
  Result := TAdoObjectFactory.create;
end;
```

Listagem 3. Interface IDBObjectFactory

```
IDBObjectFactory = interface
 functionTDBObjectAbstractFactory CreateQuery(AOwner:TComponent = nil):TDataSet;
 functionTDBObjectAbstractFactory CreateConnection(AOwner:TComponent = nil):
 TCustomConnection;
end;
```

caso da Query, além de criarmos o objeto Query da tecnologia, nós já atribuímos a ele a connection usada.

Veja que as classes implementam a interface IDBObjectFactory, que é quem define os métodos CreateConnection e CreateQuery. É neste ponto que criamos as classes concretas de acesso a banco de dados. Note que o código de ambas as classes é praticamente idêntico, justamente por implementarem a mesma interface, a diferença fica por conta da criação de um TSQLConnection ou TADOConnection para as conexões, ou TSQLQuery ou TADOQuery para os DataSets.

A herança de TInterfacedObject se faz necessária para que não seja preciso implementar os métodos básicos definidos pela interface base de IDBObjectFactory, por padrão, IInterface. Essa interface do Delphi se encarrega de implementar contadores de referência para liberação automática de objetos da memória.

Adapter genérico

Na **Listagem 6** temos a nossa classe genérica da adaptação. Nela temos um atributo FDataSet, que será o nosso objeto adaptado. Esta classe já possui alguns métodos implementados, pois possuem o mesmo comportamento tanto no dbExpress quanto no ADO, como por exemplo Open e Close. Outros métodos são diferentes e por isso são declarados como abstratos para que sejam implementados nos adapters específicos.

O código principal da classe encontra-se no construtor. Nele armazenamos uma referência ao Owner (proprietário) para ser utilizado futuramente na construção dos adapters concretos. A seguir, usamos nossa fábrica abstrata para criar objetos concretos do tipo TSQLQuery ou TADOQuery. A seguir, criamos uma StringList para armazenar a propriedade SQL.

Adapter para dbExpress

Na **Listagem 7** vemos a criação do adapter para dbExpress, retornando uma TSQLQuery no método InstanceDataSet. Temos também o método ChangeSql para atualizar o SQL do DataSet. Nós não implementamos o ChangeSql no adapter genérico pois TDataSet não possui nenhuma propriedade para armazenar esta informação.

O método SetParamValue possui implementação diferente nos dois, pois no caso do dbExpress temos a propriedade Params para acessar os parâmetros enquanto que no ADO o acesso é pela propriedade Parameters.

Nota

Como a implementação da classe de adaptação para ADO (TADOQueryAdapter) é muito parecida com a do dbExpress , a mesma foi omitida, ela pode ser encontrada no endereço para download deste artigo, juntamente com o código completo deste exemplo.

Fábrica de Adapters

Na **Listagem 8** temos nossa fábrica de adapters, que será usada pela DAO para retornar o adapter que deverá ser usado na aplicação de acordo com o tipo de conexão utilizada. Ela possui um único método estático, chamado GetAdapter, que retorna uma instância concreta de um adapter (dbExpress ou ADO).

Listagem 4. Implementação da factory para dbExpress

```
...
 TDbxObjectFactory = class(TInterfacedObject,IDBObjectFactory)
 private
  FInstance:TDbxObjectFactory;
 public
  function CreateConnection(AOwner: TComponent = nil): TCustomConnection;
  function CreateQuery(AOwner: TComponent = nil): TDataSet;
  constructor create();
 end;
...
function TDbxObjectFactory.CreateConnection(
 AOwner: TComponent): TCustomConnection;
begin
 result := TSQLConnection.Create(AOwner);
end;
function TDbxObjectFactory.CreateQuery(AOwner: TComponent): TDataSet;
begin
 result := TSQLQuery.Create(AOwner);
 TSQLQuery(result).SQLConnection := TSQLConnection(TConexaoBD.GetInstance.ConexaoBD);
end;
```

Listagem 5. Implementação da factory para ADO

```
...
type
 TAdoObjectFactory = class(TInterfacedObject,IDBObjectFactory)
 public
  function CreateConnection(AOwner: TComponent = nil): TCustomConnection;
  function CreateQuery(AOwner: TComponent = nil): TDataSet;
 end;
...
function TAdoObjectFactory.CreateConnection(
 AOwner: TComponent): TCustomConnection;
begin
 result := TADOConnection.Create(AOwner);
end;
function TAdoObjectFactory.CreateQuery(AOwner: TComponent): TDataSet;
begin
 result := TADOQuery.Create(AOwner);
 TADOQuery(result).Connection := TADOConnection(TConexaoBD.GetInstance.ConexaoBD);
end;
```

DAOs

Agora precisamos redefinir nossa TClienteDAO, que é a classe que se encarrega de manipular os dados da tabela CLIENTE no banco de dados. A principal e única diferença com relação à implementação feita no artigo anterior, é que agora passamos a usar um Adapter ao invés de uma classe concreta. Mais exatamente, ao invés de instanciarmos uma TSQLQuery, por exemplo, para executar um comando no banco, vamos instanciar um TAdapterFactory e chamar seu método GetAdapter, o qual retornará a classe correspondente à tecnologia utilizada.

Por exemplo, na primeira versão dessa DAO, tínhamos:

```
Qry := TSQLQuery.Create(nil);
```

Agora temos:

```
Qry := TAdapterFactory.GetAdapter(nil);
```

Na **Listagem 9** podemos ver parte da implementação dos métodos de persistência de um Cliente, porém agora com referência à TSQLQueryAdapter, retornada através da factory TAdapterFactory. Dessa forma nossa DAO não conhece a tecnologia concreta de acesso a dados, pois ela faz uso de nossos adaptadores.

Listagem 6. Implementação do adapter genérico

```
...
 //Classe que será conhecida pela nossa aplicação
 //Contém os métodos que nossa aplicação acessará
 TSQLQueryAdapter = class
 Protected
  FSql: TStringList;
  FActive: Boolean;
  //Instância do DataSet usado
  FDataSet:TDataSet;
  FOwner:TComponent;
  procedure SetActive(const Value: Boolean);
  procedure SetSql(const Value: TStringList);
  function GetEof: Boolean;
 published
  property Sql:TStringList read FSql write SetSql;
  property Active:Boolean read FActive write SetActive;
  property Eof:Boolean read GetEof;
 public
  //Métodos virtuais pois possuem implementação nesta classe
  //e podem ser sobrescritos em classes filhas
  //caso algum deles seja diferente para alguma outra tecnologia
  procedure Open;virtual;
  procedure Close;virtual;
  function FieldByName(FieldName:String):TField;virtual;
  procedure Next;virtual;
  procedure Prior;virtual;
  procedure First;virtual;
  procedure Last;virtual;
  //Métodos abstratos pois não é possível determinar
  //um comportamento padrão para eles
  //tendo que ser obrigatoriamente implementados
  //nas classes filhas
  procedure InstanceDataSet();virtual;abstract;
  procedure SetParamValue(ParamName : String;ParamValue:Variant);virtual;abstract;
  procedure ExecSql(ExecDirect:boolean = false);virtual;abstract;
  procedure ChangeSql(Sender:TObject);virtual;abstract;
  constructor create(AOwner:TComponent);
 end;
...
constructor TSQLQueryAdapter.create(AOwner: TComponent);
begin
 //Armazenamos o Owner para construção futura dos componentes
 FOwner := AOwner;
 //Uso da fábrica abstrata para criar uma Query de acordo com a tecnologia
 //usada na aplicação
 FDataSet := TDBObjectAbstractFactory.GetDBObjectFactory().CreateQuery(AOwner);
 //Criação da propriedade SQL
 Sql := TStringList.Create;
 //ChangeSQL aponta para o evento OnChange para que possamos atualizar a SQL
 //do DataSet sempre que a instrução for alterada
 Sql.OnChange := ChangeSql;
end;
procedure TSQLQueryAdapter.First;
begin
 FDataSet.First;
end;
// os métodos Last, Next, Open, Close, Prior e FieldByName
// foram omitidos pois fazem exatamente o mesmo papel
// do método First acima, repassando a chamada para a FDataSet interno
procedure TSQLQueryAdapter.SetActive(const Value: Boolean);
begin
 FActive := Value;
 FDataSet.Active := Value;
end;
...
```

Listagem 7. Implementação do Adapter dbExpress

```
...
 TDBXQueryAdapter = class(TSQLQueryAdapter)
 public
  procedure InstanceDataSet; override;
  procedure SetParamValue(ParamName: string; ParamValue: Variant); override;
  procedure ExecSql(ExecDirect: Boolean = False); override;
  procedure ChangeSql(Sender: TObject); override;
 end;
...
procedure TDBXQueryAdapter.ChangeSql(Sender: TObject);
begin
 inherited;
 TSQLQuery(FDataSet).SQL.Text := Sql.Text;
end;
procedure TDBXQueryAdapter.ExecSql(ExecDirect: Boolean);
begin
 inherited;
 TSQLQuery(FDataSet).ExecSQL(ExecDirect);
end;
procedure TDBXQueryAdapter.InstanceDataSet;
begin
 inherited;
 FDataSet := TSQLQuery.Create(FOwner);
end;
procedure TDBXQueryAdapter.SetParamValue(ParamName: string;ParamValue: Variant);
begin
 inherited;
 TSQLQuery(FDataSet).ParamByName(ParamName).Value := ParamValue;
end;
```

Listagem 8. Implementação da factory de Adapters

```
...
 TAdapterFactory = class
  public
   class function GetAdapter(AOwner:TComponent):TSQLQueryAdapter;
 end;
...
class function TAdapterFactory.GetAdapter(AOwner:TComponent): TSQLQueryAdapter;
begin
 if(TAplicacao.TypeConnection = tcDbx)then
   Result := TDBXQueryAdapter.create(AOwner)
 else
 if(TAplicacao.TypeConnection = tcAdo)then
   Result := TADOQueryAdapter.create(AOwner);
end;
```

Nota

A classe possui quase a mesma codificação da versão anterior, de forma que omiti o código semelhante.

Repare que agora na implementação do método Insert temos uma variável do tipo TSQLQueryAdapter ocultando assim a tecnologia utilizada no acesso a dados. Agora se mudarmos de tecnologia ou mesmo adicionarmos uma nova tecnologia de acesso a dados, nada teremos que mudar em nossas classes DAO.

Conclusão

Com isso fechamos a refatoração do nosso pequeno framework de acesso dados. Apesar de simples, vimos na prática

Listagem 9. Métodos de persistência - TClienteDao

```
...
 TClienteDao = class(TInterfacedObject,IDao)
 public
   procedure Insert(pCliente:TBaseDomain);
 ...
 end;
...
procedure TClienteDao.Insert(pCliente: TBaseDomain);
var
 Qry:TSQLQueryAdapter;
begin
 Qry := TAdapterFactory.GetAdapter(nil);
 Qry.SQL.Add('INSERT INTO CLIENTE(NOME,IDADE,BAIRRO,ID_USUARIO) VALUES');
 Qry.SQL.Add('(:pNOME,:pIDADE,:pBAIRRO,:pID_USUARIO)');
 Qry.SetParamValue('pNOME',TCliente(pCliente).Nome);
 Qry.SetParamValue('pIDADE',TCliente(pCliente).Idade);
 Qry.SetParamValue('pBAIRRO',TCliente(pCliente).Bairro);
 Qry.SetParamValue('pID_USUARIO',TCliente(pCliente).UsuarioCadastro.Id);
 Qry.ExecSQL();
 pCliente.State := sdBrowse;
end;
```

como implementar os padrões factory, singleton, abstract factory, DAO e Adapter. Comprovamos que a implementação destes patterns nos trouxe benefícios como menor acoplamento, maior coesão, menor impacto de mudanças e maior reuso de código.

Nosso objetivo tem sido exatamente mostrar a aplicabilidade de design patterns para solução de problemas reais no desenvolvimento de aplicações em Delphi, explicando como funcionam, para que servem e demonstrando de forma prática como implementar em situações em que realmente são úteis. ●

Links

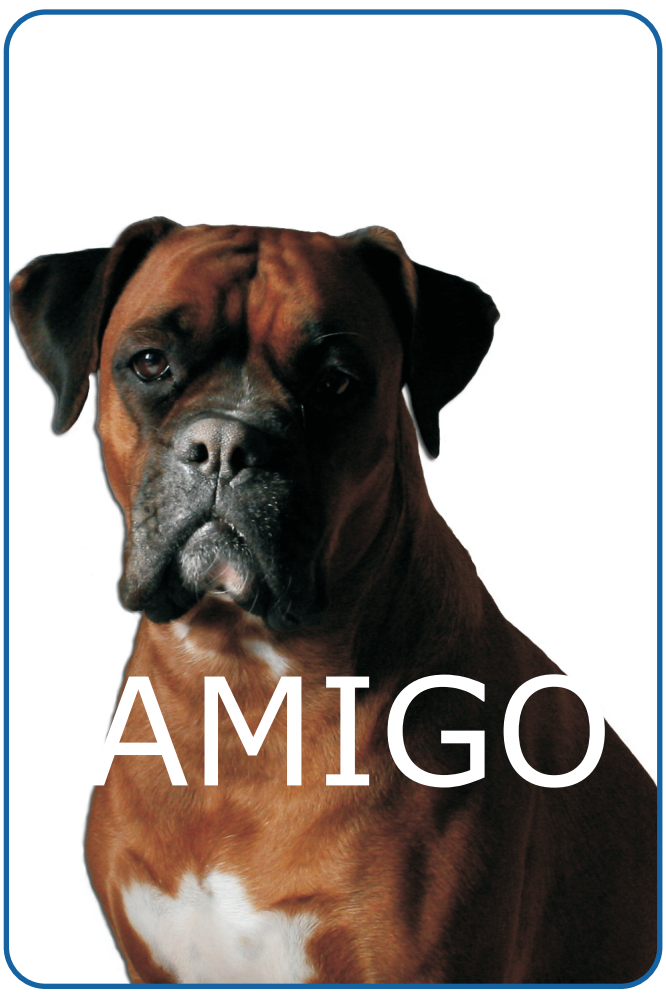
Design Patterns para o mundo real - Singleton, Factory e DAO (ClubeDelphi 115)
http://www.devmedia.com.br/post-16153-Artigo-Clube-Delphi-115-Design-Patterns-para-o-mundo-real.html

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

Para mais informações:
www.devmedia.com.br/central

DevMedia GROUP

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Primeiros passos com o novo DataSnap 2010

Conheça as novidades na prática

Fabrício Hissao Kawata
fabricio.kawata@bol.com.br
Formado em Processamento de Dados pela FATEC-TQ. Atua como Analista Programador (Delphi e Java) há 4 anos, sendo 3 destes numa empresa do ramo de telecomunicações.

Resumo do DevMan

No ano passado, com o lançamento do Delphi 2010, foi apresentada a nova versão do DataSnap, agora rebatizado como DataSnap 2010. Nesta versão, foi mantida a base da arquitetura de seu antecessor - DataSnap 2009, porém com algumas melhorias, como por exemplo o suporte a JSON, REST e conectividade utilizando o protocolo HTTP. Neste artigo vamos dar o pontapé inicial para este tipo de desenvolvimento, com a construção de uma aplicação bem introdutória.

Do que trata o artigo
Apresentar uma introdução ao novo DataSnap, lançado com o Delphi 2010, mostrando algumas de suas novidades através de um exemplo prático passo-a-passo que demonstra a construção de uma aplicação simples multicamadas (Banco de Dados, Servidor de Aplicação e Aplicação Cliente).

Para que serve
Mostrar, de forma a incentivar o público iniciante, o desenvolvimento multicamadas utilizando a tecnologia DataSnap, uma vez que o desenvolvimento deste tipo de aplicação ficou muito facilitada a partir do Delphi 2009, e que sofreu grande evolução em sua recente versão 2010.

Em que situação o tema é útil
Nos sistemas multicamadas há uma separação física das responsabilidades de um sistema, como a conectividade com o banco de dados, as regras de negócios e a interface com usuário. Desenvolver sistemas desta forma é útil por vários motivos: ganho de performance, manutenção facilitada, modularização etc.

O desenvolvimento de aplicações multicamadas teve início no Delphi 3, com o MIDAS. Esta nomenclatura permaneceu até o Delphi 6, onde foi apresentada como DataSnap. Até então, a principal característica desta arquitetura era o envolvimento com a tecnologia COM (Component Object Model) da Microsoft, para a comunicação remota (entre a camada cliente e servidora). Desde então, a cada nova versão do IDE tem-se uma evolução natural do DataSnap e foi em sua versão 2009 que houve uma grande remodelagem de sua

estrutura e sua dependência do COM totalmente removida. A conectividade ficou por conta de um novo driver dbExpress exclusivo para DataSnap. Apesar das melhorias, essa versão era restrita a conexões utilizando o protocolo TCP/IP.

A versão 2010 do DataSnap, apesar de se basear em seu antecessor, apresentou algumas novidades em relação ao DataSnap 2009:

• Agora o Delphi provê a construção desse tipo de aplicação através de dois wizards (assistentes), onde é possível a escolha entre dois grupos de aplicações: Windows (VCL tradicional, Windows Service ou Console) e Web (ISAPI, CGI ou Web App Debugger);
• Permite utilização de REST para fazer chamadas aos métodos de servidor;
• Suporte ao formato JSON;
• Função Callback, que é o modo de passar uma função como parâmetro em outra função. Ao ser completo o callback o controle volta à função original;
• Conectividade através do protocolo HTTP;
• Uso de autenticação HTTP para acesso ao Server;
• Utilização de filtros (como o de compressão de dados, que já acompanha o Delphi) e até mesmo a construção de filtros personalizados (Ex.: criptografia de dados).

Novidades na prática

Para termos uma ideia real de como funciona o desenvolvimento de um aplicativo com o novo DataSnap, iremos desenvolver uma pequena aplicação de exemplo que irá conter três camadas, conforme mostra a **Figura 1**. Tal aplicação será um sistema gerenciador de contatos pessoais que, em outras palavras, é como uma agenda de telefones. De caráter simplista, ela possuirá apenas uma tela para a manutenção dos contatos.

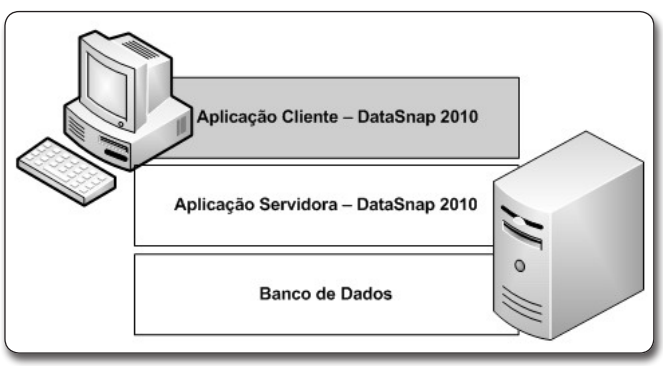


Figura 1. Arquitetura da aplicação de exemplo

Preparação do ambiente de desenvolvimento

Para iniciar o desenvolvimento, serão necessários os seguintes requisitos instalados em sua máquina (ver seção links no final do artigo):

• Delphi 2010 – será nosso IDE de desenvolvimento da aplicação DataSnap servidora e cliente Win32;
• Firebird 2.1 – será o nosso SGBD. Para sua administração será usado a ferramenta IBExpert, ambos de caráter gratuito (free).

Nota do DevMan

JSON (JavaScript Object Notation) é um formato para troca de dados baseado no objeto literal do JavaScript no formato texto, o que facilita sua interpretação e processamento por qualquer linguagem. Não obstante ao XML, o JSON constitui-se de uma coleção de propriedades e valores definidos por pares chave/valor.

Nota do DevMan

O termo REST descreve qualquer interface Web simples que utiliza XML, JSON ou texto simples e HTTP, sem as abstrações adicionais dos protocolos baseados em padrões de trocas de mensagem como o protocolo de serviços Web SOAP. No REST, cada recurso disponível no servidor é identificado por uma URL diferente.

A seguir, como sugestão crie na raiz C: uma pasta chamada DataSnap2010 e dentro desta outras três: Database, Server e Client. Estes serão os locais onde serão salvos o arquivo de banco de dados, o projeto da aplicação Server e o projeto da aplicação Client, respectivamente.

Banco de dados

Para começar o desenvolvimento de nosso exemplo, iremos fazer a construção do banco de dados da aplicação. Para isso você já deve ter instalado o Firebird e o IBExpert. A instalação de ambos não será abordada, pois está fora do escopo do artigo. Além disso, são simples e intuitivas, não sendo necessária qualquer explicação adicional. A seguir passamos para a criação efetiva do banco que será chamado de DB_CONTATOS. Para agilizar, você pode executar o script da **Listagem 1**.

Listagem 1. Script para a criação do banco de dados, tabela e demais objetos auxiliares

```
SET SQL DIALECT 3;
SET NAMES WIN1252;
SET CLIENTLIB 'fbclient.dll';

CREATE DATABASE 'C:\DataSnap2010\Database\DB_CONTATOS.fdb'
USER 'SYSDBA' PASSWORD 'masterkey'
PAGE_SIZE 4096
DEFAULT CHARACTER SET WIN1252;

CREATE SEQUENCE SEQ_CONTATOS_ID;

CREATE TABLE CONTATOS (
  ID      INTEGER NOT NULL,
  NOME    VARCHAR(80) NOT NULL,
  TELEFONE VARCHAR(15),
  DATA_CADASTRO DATE
);

ALTER TABLE CONTATOS ADD CONSTRAINT PK_CONTATOS PRIMARY KEY (ID);
```

Neste script estamos criando o banco de dados no caminho definido anteriormente, a Tabela de Contatos e um

Sequence, que nos auxiliará para fazer o autoincremento do campo ID.

Servidor

Dando início à parte prática do exemplo, o primeiro passo será construir um servidor de aplicações (Server Application), que no nosso caso será uma aplicação VCL comum. No Delphi 2010 acesse File > New > Other. Repare que em Delphi Projects existe um item chamado DataSnap Server (**Figura 2**). Nele estão contidos dois tipos de assistentes (wizards): DataSnap Server (aplicações Windows) e DataSnap WebBroker (aplicações Web). Selecione a primeira opção. A seguir é apresentada uma outra janela (**Figura 3**), onde serão configurados os últimos detalhes para que o Delphi crie o projeto DataSnap Server.

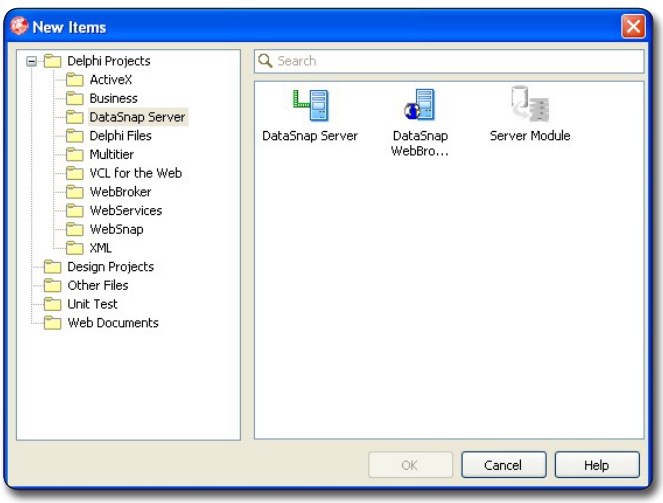


Figura 2. Localização dos wizards DataSnap Server

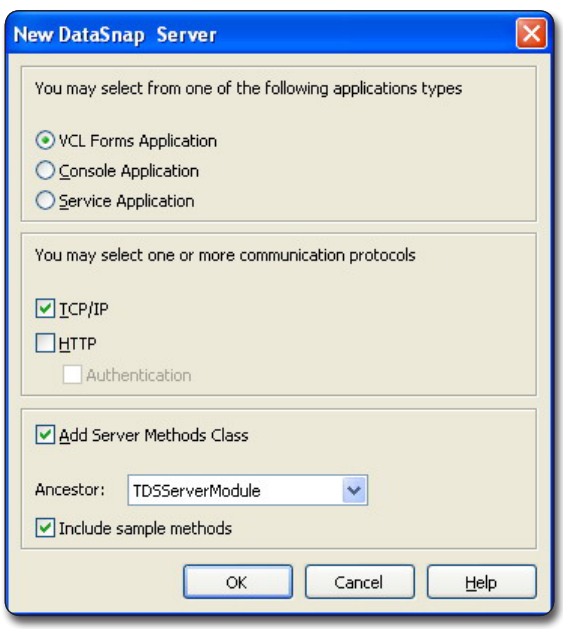


Figura 3. Configuração do projeto DataSnap a ser criado

Nesta configuração, selecionamos o tipo da aplicação (que será VCL Forms Application) e o protocolo de comunicação, que no caso será TCP/IP. Ao confirmar através do botão OK, o Delphi criará automaticamente um projeto com três arquivos: ServerContainerUnit1.pas, ServerMethodsUnit1.pas e Unit1.pas. Veremos cada um em detalhes.

A Unit1 contém um formulário vazio, que será o form principal da aplicação, portanto renomeie-o como frmMainServer, mude seu Caption para App - Server e adicione um componente TPanel (pnlServer). Para este Panel, apenas para fins visuais, altere suas propriedades Font Style para fsBold, Height para 16 e Caption para SERVER. O seu visual deve ficar parecido com a **Figura 4**.

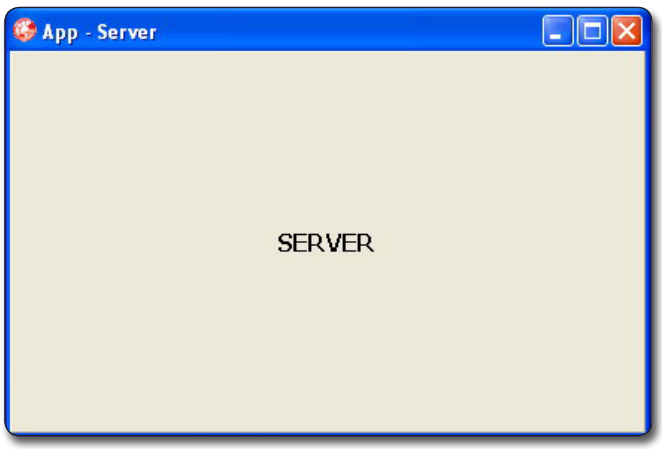


Figura 4. Formulário principal da aplicação servidora

ServerMethodsUnit1 é um ServerModule que será o container para os componentes responsáveis pelo acesso a dados do banco. Renomeie-o para SM.

ServerContainerUnit1 é um DataModule (renomeie-o para "SC") simples onde o Delphi já incluiu os três componentes principais da nova arquitetura DataSnap 2010 (**Figura 5**):

- DSServer: Componente que faz a gerência das classes do servidor (ex.: criação, transporte);
- DSServerClass: Componente que especifica a classe do servidor que será exposta aos clientes remotos para invocação de seus métodos;
- DSTCPServerTransport: Componente responsável pela configuração do protocolo de transporte TCP/IP.

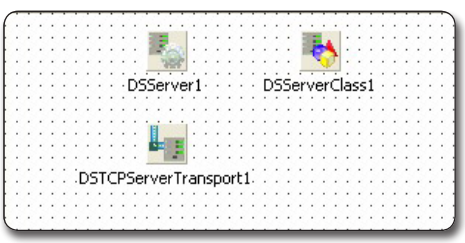


Figura 5. Componentes DataSnap

Por fim, salve tudo (Shift + Ctrl + S) na pasta Server, nomeando as units como untMainServer, untSM e untSC respectivamente e o projeto como Server. Um detalhe importante é que, como

renomeamos os objetos criados automaticamente pelo Delphi, é necessário que você renomeie no código toda e qualquer referência para eles, evitando assim erros de compilação.

O próximo passo será configurar a parte de acesso a dados da aplicação. Para tal, vá até o Data Explorer (View > Data Explorer) e dê um clique com o botão direito do mouse sobre o nó Firebird e selecione a opção Add New Connection, para que seja definida uma nova conexão. Defina o nome como DataSnap2010. A seguir, sobre o nó criado clique novamente com o botão direito e selecione a opção Modify Connection. Na tela que se abre, informe o caminho da base de dados (localhost:C:\DataSnap2010\Database\DB_CONTATOS.FDB), o User Name (SYSDBA) e o Password (masterkey). A seguir clique no botão Advanced e atribua WIN1252 para o item ServerCharSet, para que seja definido o Charset a ser usado nesta conexão. Teste a conexão e se tudo estiver OK, conforme a **Figura 6**, podemos prosseguir.

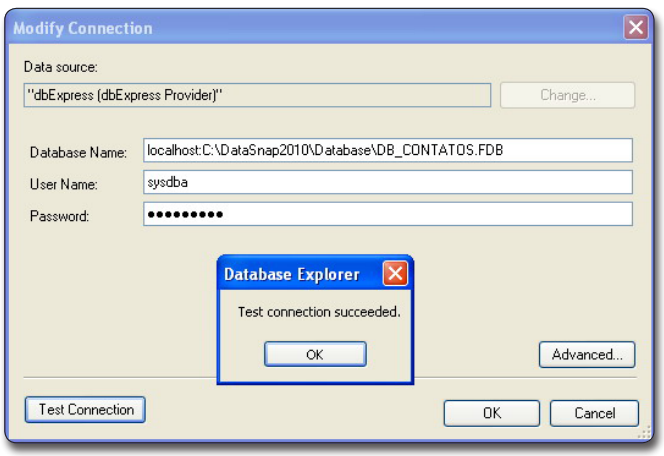


Figura 6. Teste da conexão criada

Expanda o nó que criamos e veja que em Tables já consta a tabela CONTATOS do nosso banco de dados. Arraste-a para o SM. Veja que o Delphi já insere um componente SQLConnection e um SQLDataSet, ambos configurados para a nossa conexão. Como complemento, no SQLDataSet altere seu nome para sdtContatos e sua propriedade CommandType para ctQuery, para que possamos manipular comandos SQL. Em CommandText insira o seguinte código:

```
select  ID, NOME, TELEFONE, DATA_CADASTRO from CONTATOS
where NOME like :NOME
order by NOME
```

Continuando, adicione todos os fields na lista de campos dando um duplo clique no SQLDataSet e com o botão direito selecionando Add all Fields (Ctrl + F). Na propriedade ProviderFlags de cada field, marque as flags pfInUpdate, pfInWhere e pfInKey para o campo ID, e para os demais marque apenas a primeira. Isso configura os campos de forma que o DataSnap gere corretamente as instruções de atualização. Para o campo DATA_CADASTRO, altere suas propriedades DisplayFormat para dd/MM/yyyy e EditMask para 99/99/9999, garantindo que a data a ser exibida pelo sistema esteja no padrão brasileiro.

Da paleta Data Access, arraste um DataSetProvider (dspContatos). Ligue-o ao sdtContatos através de sua propriedade DataSet. Ainda em suas propriedades, mude o UpdateMode para upWhereKeyOnly (apenas o campo chave será colocado em instruções Where geradas) e em Options marque as opções poIncFieldsProps e poPropogateChanges, para que as configurações do SQLDataSet sejam repassadas ao ClientDataSet de nossa futura aplicação cliente.

Para tornar este exemplo funcional, codifique os eventos OnCreate e OnDestroy do ServerModule (SM) conforme **Listagem 2**.

Listagem 2. Codificação dos eventos de abertura e fechamento de conexão com o banco de dados

```
procedure TSM.DSServerModuleCreate(Sender: TObject);
begin
  //conexão com o banco é aberta
  DataSnap2010.Open;
end;

procedure TSM.DSServerModuleDestroy(Sender: TObject);
begin
  //conexão com o banco é fechada
  DataSnap2010.Close;
end;
```

Nesta codificação, para o bom funcionamento da aplicação, simplesmente garantimos em momentos distintos a abertura e o fechamento da conexão com o banco de dados através do componente SQLConnection. Como você pode ter notado até aqui, nosso servidor foi construído de uma maneira muito similar ao DataSnap tradicional. A novidade ficou por conta da utilização do Wizard inicial. Dê um build no projeto (Shift + F9) e execute (F9) a aplicação servidora para deixá-la ativa e disponível para nossa aplicação cliente que desenvolveremos a seguir.

Nota

Neste momento, caso sua máquina possua um firewall, irá surgir uma mensagem em tela perguntando se você deseja desbloquear a porta a ser usada pela aplicação servidora, apenas confirme.

Cliente

Com o nosso servidor de aplicações construído e já rodando, iniciaremos agora o desenvolvimento da aplicação cliente, que será do tipo Win32. Ainda com o IDE do Delphi aberto, adicione um novo projeto VCL ao grupo de aplicações utilizando o Project Manager, como você pode ver na **Figura 7**.

Salve o projeto como Client.dpr, o formulário como untMainClient (renomei-o como frmMainClient) e o grupo de projetos como DataSnap2010Group.groupproj (salvar na pasta raiz DataSnap2010). Com nosso novo projeto aberto adicione um Data Module (File > New > Other > Delphi Files), nomeando-o como

DM e salve-o como untDM. Nele, adicione um SQLConnection (paleta dbExpress) com sua propriedade Driver apontada para o driver DataSnap, LoginPrompt marcada como false e seu Name como conConnection. Além disso, ele tem que estar configurado para a porta TCP/IP disponibilizada pelo nosso servidor de aplicações (configurada no componente DSTCP-ServerTransport1). Como na ocasião deixamos o valor como 211 para a porta, que é o valor padrão sugerido pelo Delphi, não precisaremos alterá-la aqui também. Mude para True a propriedade Connected, apenas para certificar que está tudo correto com a conexão à aplicação servidora.

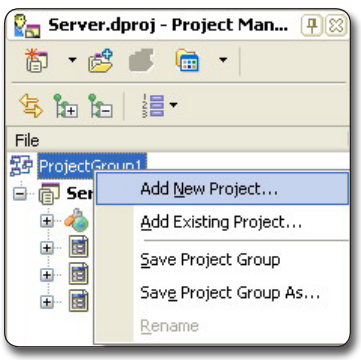


Figura 7. Project Manager

Ainda no Data Module, adicione agora um componente DS-ProviderConnection (paleta DataSnap Client) que será usado para fazer referência a uma classe do servidor, através de sua propriedade ServerClassName. Portanto, em sua propriedade SQLConnection, aponte para o nosso conConnection e em ServerClassName digite TSM, que nada mais é que o nome da classe que expomos no servidor e que queremos acessar pelo cliente.

Continuando, adicione um ClientDataSet (paleta Data Access) e renomei-o como cdsContatos. Ele será o responsável por buscar e atualizar informações de um DataSet remoto. Para configurá-lo, ligue-o ao componente DSProviderConnection1 através de sua propriedade RemoteServer. Ao fazer isto, perceba que em ProviderName já é carregada a lista de providers disponíveis no nosso servidor, que neste caso é o dspContatos. Ainda no cdsContatos, clicando com o botão direito sobre ele, escolha a opção Fetch Params para que seja mapeado no ClientDataSet os parâmetros definidos no SQLDataSet da aplicação servidora. Por fim, dê um duplo-clique sobre ele e com o botão direito acesse seu menu de contexto escolhendo Add All Fields (Ctrl +F), para que sejam adicionados todos os campos mapeados do servidor na lista de Fields do ClientDataSet. Agora no código, na seção public, defina duas funções e codifique-as conforme a **Listagem 3**.

Veja que no método ApplyContatos, chamamos o ApplyUpdates do ClientDataSet cdsContatos para que as alterações feitas em memória sejam salvas efetivamente no banco de dados. Um valor booleano é retornado indicando o sucesso ou não da operação. Já em SearchContatos é efetuada uma consulta parametrizada à tabela de Contatos do banco, para isto, fechamos o cdsContatos, passamos o parâmetro juntamente com caracteres coringas para aumentar a abrangência da pesquisa e por fim abrimos o ClientDataSet. O método também retorna um booleano, indicando se houve ou não incidências de registros.

Listagem 3. Codificação dos métodos definidos no DataModule

```
public
 { Public declarations }
 function ApplyContatos: Boolean;
 function SearchContatos(aNome: string): Boolean;
end;
...
function TDM.ApplyContatos: Boolean;
begin
 //chama método ApplyUpdates do ClientDataSet e atribui seu retorno ao Result da função
 Result:= cdsContatos.ApplyUpdates(0) = 0;
end;

function TDM.SearchContatos(aNome: string): Boolean;
begin
 //realiza a consulta através do valor parâmetro passado à função e com auxílio de caracteres
   coringas
 cdsContatos.Close;
 cdsContatos.Params[0].AsString:='%' + aNome + '%';
 cdsContatos.Open;

 //indica se houve ou não retorno de registros
 Result:= not cdsContatos.IsEmpty;
end;
```

Com nosso Data Module pronto, insira mais um form ao projeto (New>Form), nomeando-o como frmContatos e salvando-o como untContatos. Mude suas propriedades Caption para Contatos, Position para poMainFormCenter e BordeStyle para bsDialog. Por fim, adicione os seguintes componentes para que a aparência da tela fique como o da **Figura 8**:

- 1 DBGrid (dbgContatos) - paleta Data Controls;
- 1 DataSource (dsrContatos) – Data Access;
- 1 GroupBox (gbxBotoes) – Standard;
- 2 Buttons (btnBuscar, btnApply) – Standard;
- 1 DBNavigator (dbnNav) – Data Controls;
- 1 Edit (edtBuscar) – Standard;
- 4 Labels – Standard;
- 2 DBTexts – Data Controls;
- 2 DBEdits – Data Controls.

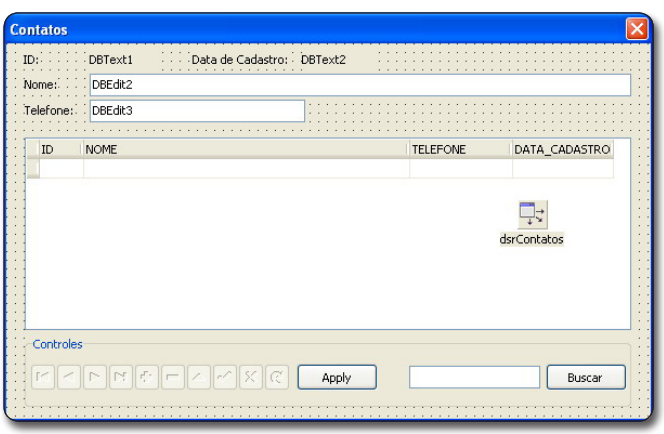


Figura 8. Aparência do formulário de contatos da aplicação cliente

O próximo passo é ligarmos o DataSource (dsrContatos) ao nosso ClientDataSet. Para isso, através do atalho Alt+F11 selecione a unit do DataModule para que esta esteja disponível para o formulário. A seguir ligue o ClientDataSet ao DataSource através de sua propriedade DataSet. Para os componentes DBGrid e DBNavigator, aponte sua propriedade DataSource ao dsrContatos. Ainda no DBGrid, dê um duplo-clique sobre ele para que seja aberto seu editor de colunas. A seguir, dê um clique com o botão direito do mouse para que seja mostrado seu menu de contexto. Selecione Add All Fields para que sejam adicionados todos os fields do DataSet. Para os controles DBEdit e DBText altere seu DataField para o field relativo no DataSet.

Codifique os botões Open e Apply e dos eventos OnClose e OnShow do formulário, respectivamente, conforme a **Listagem 4**.

Listagem 4. Codificação dos botões do formulário de Contatos

```
procedure TfrmContatos.btnBuscarClick(Sender: TObject);
begin
 //verifica se há algo digitado na caixa de texto
 if (Trim(edtBuscar.Text) <> '') then
 begin
  //executa a função de pesquisa definida no DM, passando como parâmetro o valor do Edit
  if not DM.SearchContatos(Trim(edtBuscar.Text)) then
   //caso a função não retorne registro, é exibida uma mensagem
   Application.MessageBox('Busca sem resultados!', 'Atenção', MB_ICONWARNING);
 end;
end;

procedure TfrmContatos.btnApplyClick(Sender: TObject);
begin
 //chama método ApplyContatos definido no DM, e de acordo com seu resultado emite mensagem
 if DM.ApplyContatos then
  Application.MessageBox('Erro ao salvar registro', 'Erro', MB_ICONERROR)
 else
  Application.MessageBox('Registro salvo com sucesso!', 'Sucesso', MB_ICONWARNING);
end;

procedure TfrmContatos.FormClose(Sender: TObject; var Action: TCloseAction);
begin
 //ao fechar o formulário, certifica-se do fechamento do DataSet
 dsrContatos.DataSet.Close;
end;

procedure TfrmContatos.FormShow(Sender: TObject);
begin
 //ao mostar o formulário, abre o DataSet vazio, ficando habilitadas a operação
 //de inserção
 dsrContatos.DataSet.Open;
end;
```

Nesta codificação, para o Click do botão Buscar primeiramente realizamos um teste para verificar se a caixa de entrada edtBuscar está preenchida. Se sim, é feita uma chamada ao método SearchContatos definido no Data Module DM. Caso seu retorno seja false, é emitida uma mensagem de que a busca não obteve resultados. Já para o Click do botão Apply, fazemos novamente uma chamada a um método definido no DM, o ApplyContatos. De acordo com seu retorno é emitida uma mensagem de sucesso ou erro. No evento Close do formulário, apenas certificamos o fechamento do DataSet, evitando que dados fiquem em memória desnecessariamente. Já no evento Show, garantimos que o DataSet seja aberto, habilitando dessa forma a operação de inserção de um novo registro.

Para finalizar essa primeira parte da aplicação cliente, volte ao formulário frmMainClient e altere suas propriedades Caption para App – Client e WindowState para wsMaximized. Adicione um componente MainMenu da paleta Standard. Com um duplo-clique sobre este, adicione um item com o Caption Manutenção e um subitem Contatos. Através do atalho Alt + F11, dê uses em frmContatos, para que possamos chamar este formulário através do menu. Para tal, novamente com um duplo clique sobre o MainMenu acesse o evento OnClick do item Contatos e codifique-o conforme a **Listagem 5**.

Listagem 5. Codificação do item de menu

```
procedure TfrmMainClient.Contatos1Click(Sender: TObject);
begin
 try
  //faz a instanciação do formulário de Contatos
  Application.CreateForm(TfrmContatos, frmContatos);

  //exibe o formulário em tela através do ShowModal
  frmContatos.ShowModal;
 finally
  //após seu uso, libera o formulário da memória
  FreeAndNil(frmContatos);
 end;
end;
```

Nesta codificação, utilizamos o método CreateForm de Application para fazer a instância do formulário frmContatos para em seguida mostrá-lo em tela através de seu método ShowModal. Quando este não for mais usado, é liberado da memória.

Implementando métodos no servidor

Faremos agora a implementação de alguns métodos no servidor que serão consumidos pela aplicação cliente. Primeiramente finalize a aplicação Server e volte ao seu projeto. Na unit do ServerModule (untSM) adicione dois métodos na seção public e implemente-os como na **Listagem 6**.

Na **Listagem 6**, o método GenerateID recebe como parâmetro o nome de um objeto Sequence do banco de dados Firebird e apresenta como retorno o valor deste (do tipo inteiro), e que será usado em campos autoincremento. Para isto, no corpo do método instanciamos um objeto do tipo SQLQuery que executará um comando SQL no banco de dados, especificamente para gerar o valor da Sequence. A seguir, o Result da função recebe este valor, e o objeto criado é liberado da memória. Já para o outro método, chamado de GetDateServer, seu Result retornará a Data definida no Servidor, do tipo TDateTime, e será usada como uma forma de padronizar a data de cadastro dos registros no Cliente.

Faça novamente um build da aplicação (Shift+F9) e execute-a, deixando o servidor ativo novamente. Volte ao

projeto da aplicação cliente. Para fazermos a chamada a esse método que acabamos de implementar existem basicamente duas maneiras: utilizar o componente SQLServerMethod da paleta dbExpress, fazendo a chamada ao método do servidor como se fosse uma stored procedure (inclusive podendo fazer uso de seus parâmetros de entrada e saída) ou gerar uma classe proxy na aplicação cliente, utilizando esta para fazer a chamada.

Listagem 6. Exemplo de método servidor

```
public
 { Public declarations }
 function GenerateID(aSequence: string): Integer;
 function GetDateServer: TDateTime;
end;

{função que retorna um ID sequencial}
function TSM.GenerateID(aSequence: string): Integer;
var
 Qry: TSQLQuery;
begin
 try
  //instancia objeto em memória
  Qry:= TSQLQuery.Create(Self);

  //atribui o Connection à sua propriedade
  Qry.SQLConnection:= DataSnap2010;

  //atribui comando SQL para execução do Sequence
  Qry.SQL.Add('SELECT NEXT VALUE FOR ' + aSequence + ' from RDB$DATABASE');

  //executa o comando SQL
  Qry.Open;

  //retorna o ID
  Result:= Qry.Fields[0].AsInteger;
 finally
  //liberda objeto criado da memória
  FreeAndNil(Qry);
 end;
end;

function TSM.GetDateServer: TDateTime;
begin
 {função que retorna a data do servidor}
 Result:= Now;
end;
```

Nota

Desde a versão 2009 do DataSnap as chamadas remotas a estes métodos são baseadas na RTTI (Runtime Type Information) do próprio Delphi, com suporte à passagem de parâmetros baseados nos tipos dos parâmetros do dbExpress.

Primeiramente demonstraremos o uso do componente, portanto, na Tool Palette navegue até o grupo dbExpress e adicione um SQLServerMethod (ssmMethodServer) ao DataModule. Ligue-o ao SQLConnection através de sua propriedade de mesmo nome. Repare que a propriedade ServerMethodName foi carregada com os métodos disponíveis no servidor (**Figura 9**), inclusive o que acabamos de criar além de outros métodos auxiliares. Selecione o item TSM.GenerateID.

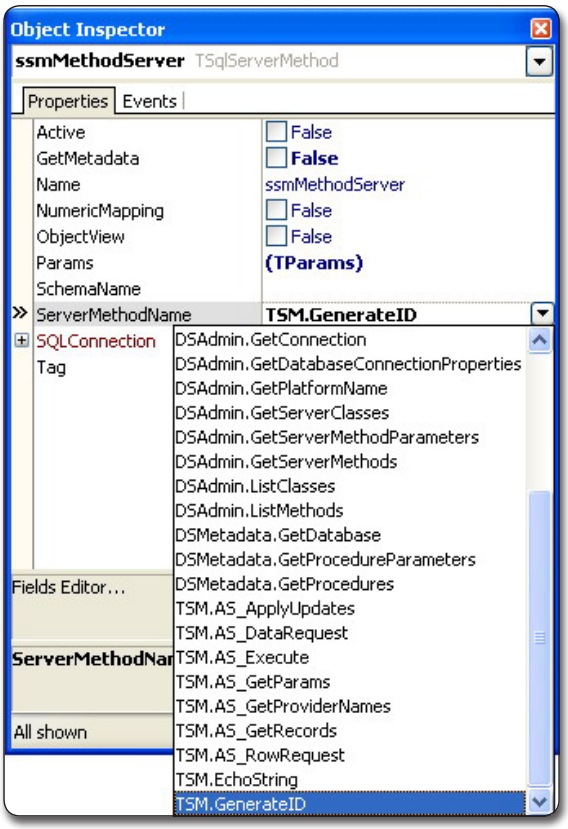


Figura 9. Lista de métodos do servidor que estão disponíveis para uso no lado cliente

Ao fazer isso, o Delphi já faz o mapeamento completo do método do servidor, inclusive trazendo os parâmetros de entrada e saída deste, conforme pode ser visto na **Figura 10**.

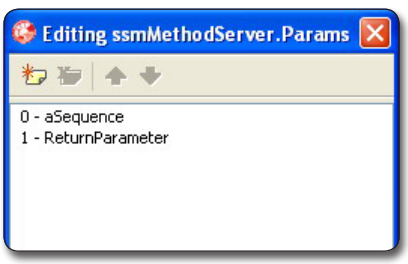


Figura 10. Parâmetros de entrada e saída do método do servidor

Utilizaremos este método para atribuir valor ao campo ID, simulando um autoincremento. Portanto, no evento OnNewRecord do cdsContatos, insira o código da **Listagem 7**. Utilizamos um código bem simples, que irá chamar o método ExecuteMethod do componente SQLServerMethod e atribuir o seu retorno ao campo ID.

Da mesma forma, podemos utilizar uma classe proxy cliente para fazer a chamada a este ou qualquer outro método disponibilizado pelo servidor. Portanto, ainda no Data Module da aplicação cliente, dê um clique de direita sobre o componente SQLConnection (conConnection), no menu suspenso que se abre, selecione a opção Generate DataSnap client classes, conforme mostra a **Figura 11**.

Listagem 7. Autoincremento do campo ID

```
procedure TDM.cdsContatosNewRecord(DataSet: TDataSet);
begin
 //passa parâmetro(Nome do Sequence) para o método remoto
 ssmMethodServer.Params[0].AsString:='SEQ_CONTATOS_ID';

 //executa método remoto do servidor
 ssmMethodServer.ExecuteMethod;

 //atribui o ID gerado ao campo ID do DataSet
 cdsContatosID.AsInteger:= ssmMethodServer.Params[1].AsInteger;
end;
```

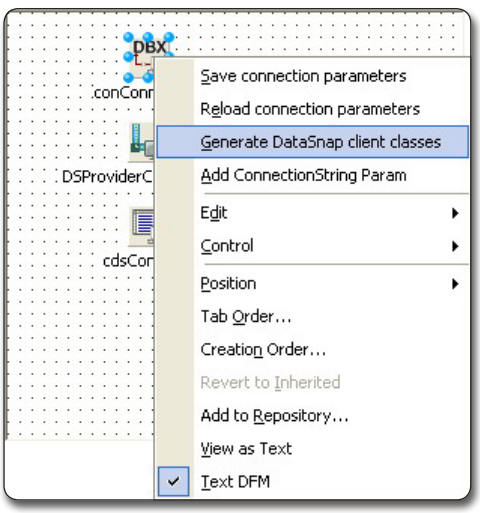


Figura 11. SQLConnection - Opção para gerar classe proxy cliente

Veja que o Delphi nos gerou automaticamente uma unit com a classe que podemos usar para chamar os métodos do servidor. Em outras palavras, ela fará o mesmo processo do componente SQLServerMethod, via código, ao invés do uso do componente. Salve esta unit como untProxyClient.

Ainda no Data Module, iremos utilizar o método GetDateServer da classe Proxy para atribuirmos ao campo DATA_CADASTRO a data do servidor, simulando um ambiente onde a aplicação cliente esteja numa máquina diferente da aplicação servidora, porém onde está se utilizando uma data centralizada a todas estações. Portanto, no evento OnBeforePost do cdsContatos adicione o código da **Listagem 8**. Para fazer uso da classe proxy, é necessário dar um uses em sua unit untProxyClient, através do atalho Alt+F11 ou da opção File > Use Unit.

Neste código, é criada uma instância da classe proxy cliente que geramos através do SQLConnection. Em seguida, fazemos através dela a chamada a um método remoto que, neste caso, é o GetDateServer, que será usado para atribuir um valor ao campo DATA_CADASTRO do ClientDataSet. Ao final, a instância criada é liberada da memória.

Realização dos testes

Com o desenvolvimento do nosso projeto exemplo finalizado, realize alguns testes para comprovar o funcionamento das aplicações cliente e servidora, além das funções definidas nestas. Como primeiro passo, certifique-se que a aplicação Server esteja rodando. A seguir, execute a aplicação Client e através de seu menu abra a tela de manutenção de Contatos. Faça algumas inserções, consultas e alterações e veja que tudo funciona de forma similar a uma aplicação Client/Server tradicional.

Listagem 8. Chamando método remoto pela classe proxy cliente

```
procedure TDM.cdsContatosBeforePost(DataSet: TDataSet);
var
 ClientClass: TSMClient; //variável do tipo da classe proxy cliente
begin
 try
  //cria uma instância da classe proxy
  ClientClass:= TSMClient.Create(DM.conConnection.DBXConnection);

  //faz a chamada do método remoto, atribuindo seu retorno ao campo DATA_CADASTRO
  cdsContatosDATA_CADASTRO.AsDateTime:= ClientClass.GetDateServer;
 finally
  //libera da memória
  FreeAndNil(ClientClass);
 end;
end;
```

Nota do DevMan

Anteriormente à versão 2009 do DataSnap, a comunicação entre aplicação servidora e cliente se dava através de uma tecnologia da Microsoft denominada COM, utilizando-se interfaces. Já no DataSnap 2009, a dependência do COM foi totalmente removida. A comunicação entre as aplicações passou a ser feita diretamente sobre o protocolo TCP/IP e posteriormente sobre TCP/IP e HTTP na versão 2010.

Nota

No novo DataSnap não há a necessidade de se registrar o servidor nem qualquer outro tipo de serviço, como acontecia nas versões anteriores a 2009.

Conclusão

Vimos neste artigo como está facilitado o desenvolvimento de aplicações multicamadas utilizando-se o novo DataSnap 2010. Este tema que era interpretado como algo muito complexo pelos desenvolvedores iniciantes e menos experientes, agora passa a ser um assunto tentador, visto a agilidade e clareza de desenvolvimento. Essa facilidade ficou evidente desde a remodelagem do DataSnap, lançado com o Delphi 2009. ●

Links

Rad Studio 2010 - Trial
http://downloads.embarcadero.com/free/delphi

Firebird 2.1
http://www.firebirdsql.org/index.php?op=files&id=engine_213

IBExpert
http://ibexpert.net/ibe/index.php?n=Main.DownloadFree

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Mapeamento Objeto / Relacional

Aumente a elegância no desenvolvimento sem perder produtividade

Do que trata o artigo

Neste artigo abordaremos um assunto de suma importância para projetos desenvolvidos orientados a objetos, a persistência de dados. Independente da linguagem que você adotar para desenvolver seu projeto você sempre irá esbarrar em um problema: seus objetos estão na memória e você precisa persisti-los em um repositório de dados, geralmente um SGBD, então vem a questão: Como transformar meus objetos em comandos SQL? Veremos aqui uma forma simples e rápida para resolver isso.

Para que serve

O mapeamento objeto / relacional, tema do nosso artigo, serve para resolver a impedância que temos entre os modelos de negócio orientados a objetos e os bancos relacionais. Isso porque a linguagem padrão dos SGBD é SQL, sendo assim, temos que mapear nossos objetos extraindo deles suas informações para então montar a instrução e enviar para o banco de dados. Os frameworks de persistência, como o Lumine (estudado neste artigo), nos auxiliam neste processo.

Em que situação o tema é útil

A utilização de mecanismos de mapeamento objeto / relacional aumenta a produtividade e reduzem o tempo de desenvolvimento de um projeto, uma vez que o desenvolvedor foca sua atenção apenas nas regras de negócio, deixando o trabalho pesado por conta do framework. Será necessário um tempo para conhecer e se familiarizar com o framework, porém, o tempo é recuperado ao longo do desenvolvimento do projeto.

Rodrigo Carreiro Mourão
rodrigocarreiro@rmfactory.com.br
Atualmente é sócio diretor da RM Factory Soluções em TI, empresa especializada em consultoria na área de TI e desenvolvimento de softwares customizados. Possui os títulos de BDS2006 Win32 Product Certified e Borland Instructor Certified. Editor Técnico da Revista ClubeDelphi e Coordenador da área de Delphi no sistema de suporte on-line da DevMedia. Palestrante da Borland Conference 2007. Mantenedor da comunidade www.delphisophp.com, a primeira do Brasil 100% feita com Delphi for PHP.

Resumo do DevMan

Quando definimos uma solução, temos que pensar qual arquitetura utilizar, quais padrões, plataforma e no caso de persistência, qual framework de mapeamento. Para Java temos o Hibernate, em .NET temos o ADO.NET Entity Framework, NHibernate e outros. E para o PHP temos o Lumine. Neste artigo vamos entender o que são e para que servem os frameworks de persistência. Iremos conhecer o Lumine, um framework de persistência objeto relacional para PHP, 100% Free e muito intuitivo. Para demonstrar isso vamos desenvolver um pequeno exemplo totalmente orientado a objetos.

Hoje em dia as principais linguagens de programação no mercado são orientadas a objetos ou suportam orientação a objetos como alguns gostam de falar, e isso é muito bom. A orientação a objetos é uma filosofia de desenvolvimento que nos permite uma melhor organização e reaproveitamento do código. Ela também nos permite, entre outras coisas, mapear nossas entidades do mundo real para o mundo computacional. Em paralelo a este crescimento da orientação a objetos vemos também um crescimento e consolidação dos bancos de dados relacionais que têm como linguagem padrão a SQL.

Num projeto orientado a objetos temos nossas regras de negócio mapeadas, ou melhor, descritas em nossas classes. Para que possamos de fato e de verdade lançar mão destas regras, temos que instanciar objetos a partir destas classes. Uma vez tendo feito isso, nosso objeto reside na memória pronto para receber os dados enviados pelo usuário e aplicar suas regras.

Esse é um dos motivos que nos levam a trabalhar como banco de dados, armazenar nossas informações para posterior consulta. Mas se tratando de objetos não podemos esquecer que os mesmos estão instanciados na memória e não podemos simplesmente pegá-los e jogá-los em um banco de dados, até porque os SGBD relacionais utilizam SQL. É aí que surge o que muitos chamam de impedância objeto relacional, uma vez que, se não trabalharmos com banco de dados orientados objetos, não temos outro jeito de salvá-los em um SGBD relacional sem que seja mapeando o objeto para SQL, o famoso mapeamento objeto / relacional.

Mapeamento objeto / relacional implica em ler um objeto e dependendo do que se deseja fazer com ele, gerar o comando SQL correspondente, seja um Insert, Update ou Delete. Um padrão muito usado para isso é o DAO, que prega que para cada classe de negócio em nosso modelo temos que ter uma classe de persistência que irá se encarregar da comunicação com o banco de dados e gerar os comandos para a persistência. Você pode estar pensando: "Mas isso pode dar muito trabalho", e é verdade, ainda mais se em seu sistema você possuir um número muito grande de classes.

É neste contexto que os frameworks de mapeamento objeto / relacional entram. Eles fazem todo o trabalho pesado, tudo que diz respeito à persistência, conexão com banco de dados, comandos SQL, ficam por conta deles. A ideia é que o analista ou programador gaste seu tempo no mais importante, o modelo de negócios.

Neste artigo iremos trabalhar com o framework Lumine. Aprenderemos a montar um ambiente de desenvolvimento completo, instalando o PHP, MySql e Apache. Veremos onde baixar e como utilizar este framework e por fim faremos um exemplo real.

Montando o ambiente

Neste exemplo optei por trabalhar com o WAMPServer, que pode ser baixado diretamente da Web sem nenhum custo e é muito fácil de instalar. Não irei me aprofundar no assunto, pois a instalação do WAMPServer foge do escopo do nosso projeto, porém basta seguir as instruções do instalador. Na seção Links no final deste artigo você irá encontrar o endereço para baixar a versão mais recente do WAMPServer. A **Figura 1** mostra o painel de controle do WAMPServer. Após instalado, ele fica no Tray do Windows.

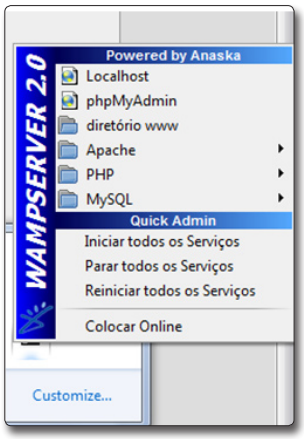


Figura 1. Painel de controle do WAMPServer

Nota do DevMan

DAO (Data Access Object) é um padrão para persistência de dados que permite separar regras de negócio das regras de acesso a banco de dados. Numa aplicação que utilize a arquitetura MVC, todas as funcionalidades de bancos de dados, tais como obter as conexões, mapear objetos para tipos de dados SQL ou executar comandos SQL, devem ser feitas por classes DAO.

Nota do DevMan

Conheça alguns frameworks populares para mapeamento objeto / relacional:

O Hibernate é um Framework para o objeto / relacional escrito na linguagem Java, mas também disponível em .NET e comumente conhecido como NHibernate. Este framework facilita o mapeamento dos atributos entre uma base tradicional de dados relacionais e o modelo objeto de uma aplicação, mediante o uso de arquivos (XML) para estabelecer esta relação.

O ADO.NET Entity Framework (EF) permite o mapeamento das tabelas do banco de dados como objetos para abstrair o acesso a dados, deixando para o desenvolvedor que já conhece orientação a objetos mais um grande mecanismo de acesso sem precisar conhecimento aprofundado de banco de dados e de SQL. Quando usamos o EF também fazemos uso da LINQ.

LINQ - Language Integrated Query - É um conjunto de recursos introduzidos no .NET Framework 3.0 que permite a realização de consultas diretamente em base de dados, documentos XML, estrutura de dados, ou coleção de objetos por exemplo, usando uma sintaxe parecida com a linguagem SQL.

ECO – Enterprise Core Objects - É um Framework para persistência de objetos, onde você trabalha com classes e objetos, fazendo a modelagem utilizando UML. Então basicamente, você faz o diagrama de classes, configura qual banco de dados será utilizado e o ECO faz toda a mágica gerando o código SQL dessas classes para o sistema funcionar.

tiOPF - Framework Open Source para Delphi e Free Pascal, que simplifica o mapeamento de um modelo de negócios orientado a objeto para um banco de dados relacional.

WAMPServer instalado, vamos então proceder com o download do pacote de script com o Lumine. Na seção Links no final deste artigo você encontrará o endereço para baixar o arquivo lumine-1.5.4.rar. Tendo feito isso vamos montar a estrutura do nosso exemplo. Se você instalou o WAMP no caminho padrão, então ele deve ter criado a pasta C:\wamp. Dentro deste diretório temos uma pasta chamada apps, que inclusive já possui o PHPMyAdmin. Vamos criar dentro deste diretório uma pasta chamada exemplolumine, nela iremos descompactar o conteúdo do arquivo lumine-1.5.4.rar. Por questão de organização descompacte o arquivo em uma pasta chamada lumine. A **Figura 2** mostra como deverá ser a estrutura de pastas.

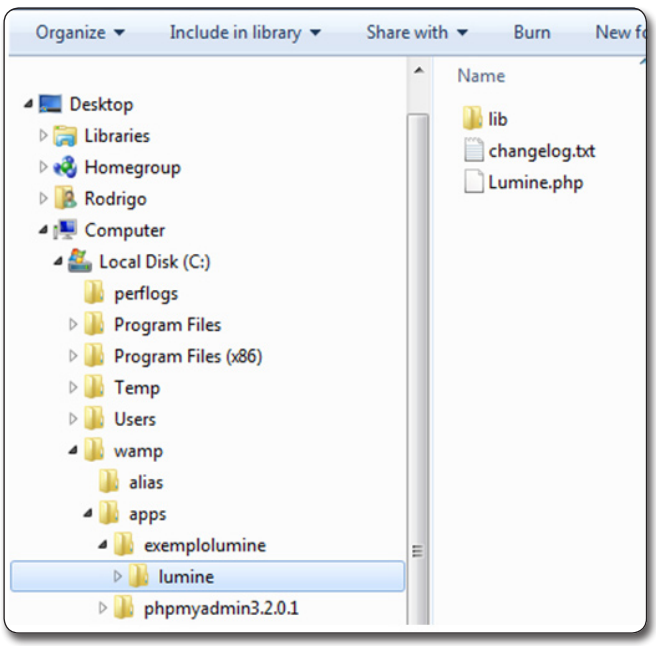


Figura 2. Estrutura de pastas

Configuração do Lumine

O arquivo lumine-conf.php contém as principais configurações do framework Lumine. Este arquivo deverá conter um array com os parâmetros necessários para o Lumine trabalhar, tais como host, username, password da base dados entre outros parâmetros. Observe o código da **Listagem 1**, ele mostra um exemplo do arquivo lumine-conf.php. A **Tabela 1** contém a descrição para os principais parâmetros.

Criando a base de dados e arquivo de configuração

Para que possamos criar nosso primeiro exemplo, vamos criar um banco de dados MySQL com apenas uma tabela simples. Feito isso vamos utilizar o Lumine para mapear nossos objetos e persistir nesta base dados. A **Listagem 2** mostra o script de criação do BD. Você poderá utilizar o próprio phpMyAdmin para poder gerar um banco de dados através deste script ou então utilizar a ferramenta de sua preferência. Não entraremos em detalhes de como criar a base de dados nem como criar a tabela, pois isso foge ao escopo do artigo, porém esta é uma tarefa extremamente simples.

Listagem 1. Exemplo de arquivo de configuração

```
<?php
$lumineConfig = array(
  'dialect' => 'MySQL',
  'database' => 'db_exemplo',
  'user' => 'root',
  'password' => '',
  'port' => '3306',
  'host' => 'localhost',
  'class_path' => dirname(__FILE__),
  'package' => 'class',
  'options' => array(
    'schema_name' => '',
    'generate_files' => '1',
    'generate_zip' => '',
    'class_sufix' => '',
    'remove_count_chars_start' => '',
    'remove_count_chars_end' => '',
    'remove_prefix' => 'tbl_',
    'create_entities_for_many_to_many' => '',
    'plural' => 's',
    'create_controls' => 'BasicForm',
  )
);
?>
```

Listagem 2. Script de criação da base de dados

```
DROP DATABASE 'db_exemplo';
CREATE DATABASE 'db_exemplo';
USE 'db_exemplo';

DROP TABLE IF EXISTS 'tbl_contato';
CREATE TABLE IF NOT EXISTS 'tbl_contato' (
 'contatoid' int(11) NOT NULL AUTO_INCREMENT,
 'Nome_Contato' varchar(50) NOT NULL,
 'Email_Contato' varchar(255) NOT NULL,
 'Telefone_Contato' varchar(15) NOT NULL,
 PRIMARY KEY ('contatoid')
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

Uma vez tendo criado o banco de dados e a tabela, passemos então à criação do arquivo de configuração. Abra o bloco de notas e coloque nele o código que já vimos na **Listagem 1**. Salve este arquivo como lumine-conf.php na nossa pasta exemplolumine que criamos dentro do diretório apps mostrado na **Figura 2**.

Nota

Repare que propositalmente no script da **Listagem 2** o nome da tabela de contato é tbl_contato, pois no nosso arquivo de configuração temos no parâmetro remove_prefix a palavra tbl_.

Nós já poderíamos começar a criar nossas classes, porém, há uma boa prática quando desenvolvemos com o Lumine. Ela prega que devemos ter um arquivo de configuração básico, o config.php. Isso facilita a organização de nossa aplicação, separando cada configuração em um arquivo específico fora do arquivo lumine-conf.php. Além disso, usando um arquivo config.php podemos facilmente reaproveitar o arquivo lumine-conf.php para outras aplicações e até mesmo ter mais de um arquivo de configuração e, através do arquivo config.php, decidirmos qual arquivo utilizar. Crie então um novo arquivo no bloco de notas e nele coloque o código da **Listagem 3**.

Parâmetro	Descrição
dialect	Tipo de banco de dados usado na aplicação. O Lumine trabalha com MySql, PostgreSQL e Firebird.
database	Nome da base de dados que será utilizada na aplicação.
user	Nome de usuário que será utilizado para conectar na base de dados.
password	Senha para acesso a base de dados utilizada na aplicação.
port	Porta utilizada para conexão à base de dados.
host	Endereço do servidor onde está hospedado ou instalado o banco de dados.
class_path	Nele você deverá informar o caminho onde estão os arquivos da sua aplicação.
package	No Lumine você pode colocar suas classes de negócio em um pacote, que nada mais é que uma pasta.
options	Nesta seção ficam os parâmetros opcionais de configuração. A maioria deles se refere a parâmetros para engenharia reversa (assunto não abordado neste artigo).
schema_name	Este parâmetro é utilizado para informar ao Lumine o nome do esquema que iremos utilizar (no caso do PostgreSQL), sendo assim somente é aplicado quando trabalhamos com este banco.
generate_files	Este parâmetro indica ao Lumine se ele deve ou não gerar os arquivos diretamente na pasta de destino em caso de engenharia reversa.
generate_zip	Quando alterado para 1 este parâmetro faz com que o Lumine gere um arquivo zip na pasta de destino contendo todos os arquivos gerados pela engenharia reversa.
class_sufix	Caso deseje, você poderá informar um sufixo para seus arquivos das classes que serão geradas pela engenharia reversa.
create_entities_for_many_to_many	Se em seu modelo você possuir tabelas para relacionamentos muitos para muitos, basta passar 1 para este parâmetro para que o Lumine gere classes para estas tabelas. O padrão é não gerar.

Tabela 1. Descrição dos principais parâmetros

Listagem 3. Arquivo config.php

```
<?php
require_once 'lumine/Lumine.php';
require_once dirname(__FILE__).'/lumine-conf.php';
$cfg = new Lumine_Configuration($lumineConfig);
date_default_timezone_set('America/Sao_paulo');
spl_autoload_register(array('Lumine','Import'));
register_shutdown_function(array($cfg->getConnection(),'close'));
?>
```

O script criado na **Listagem 3** é muito simples. Primeiro carregamos o arquivo lumine.php que contém todas as classes necessárias para o Lumine entrar em ação. Feito isso carregamos também o arquivo de configuração do Lumine criado na **Listagem 1**. Repare que com o uso do arquivo config.php podemos montar a estrutura de pasta que quisermos, basta que aqui nós carreguemos o arquivo do lugar correto.

Na sequência criamos uma variável $cgf e instanciamos a classe de configuração do Lumine, passando como parâmetro o array $lumineConfig criado na **Listagem 1**. As últimas quatro linhas são configurações extras onde, pela ordem, definimos o timezone padrão, habilitamos o autocarregamento das classes do Lumine e por fim registramos uma função para que a conexão com o banco seja fechada ao final dos scripts.

Criando o mapeamento

Vamos então criar nossa primeira classe de negócio com o Lumine. Vale lembrar que no arquivo de configuração criado na **Listagem 1** nós definimos que nossas classes de negócio ficariam no package class, então no diretório exemplolumine vamos criar uma pasta chamada class. Crie o arquivo da **Listagem 4** e salve o arquivo como contato.php dentro da pasta class.

Listagem 4. Classe Contato

```
01 <?php
02 class Contato extends Lumine_Base {
03 protected $_tablename = 'tbl_contato';
04 protected $_package  = 'class';
05 public $codigo;
06 public $nome;
07 public $email;
08 public $telefone;
09 protected function _initialize()
10 {
11 $this->_addField("codigo", "contatoid", "int", 11, array('primary' =>  true, 'notnull' =>
   true, 'autoincrement' => true));
12 $this->_addField("nome", "Nome_Contato", "varchar", 50,  array('notnull' => true));
13 $this->_addField("email", "Email_Contato", "varchar", 255,  array('notnull' => true));
14 $this->_addField("telefone", "Telefone_Contato", "varchar", 15,  array('notnull' => true));
15 }
16 public static function staticGet($pk, $pkValue = null)
17 {
18 $obj = new Contato;
19 $obj->get($pk, $pkValue);
20 return $obj;
21 }
22 function __destruct()
23 {
24 parent::__destruct();
25 }
26 }
27 ?>
```

Quando criamos uma classe de negócio que será persistida com o Lumine temos que seguir alguns passos. Primeiro devemos herdar de Lumine_Base (linha 2). O segundo passo é definir qual o nome da tabela para o qual a classe será mapeada (linha 3). Repare que informamos para a propriedade $_tablename o nome da tabela que é mapeada pela classe: tbl_contato. O terceiro passo é informar em qual package está a classe em questão, no nosso caso nossas classes estão no package class e isso nós informamos na linha 4. O quarto passo é sobrescrever o método _initialize, que serve para configurar o mapeamento objeto / relacional propriamente dito.

No início o Lumine utilizava arquivos XML para fazer o mapeamento da classe / tabela, assim como é no Hibernate, porém nas versões mais recentes do Lumine este mapeamento é feito dentro da própria classe. Isso é feito a partir da linha 9 e vai até a Linha 15. Basicamente o que temos que fazer é informar ao método _addField que está na classe Lumine_Base para adicionarmos o mapeamento para cada propriedade em nossa classe. A sintaxe é a seguinte: _addField(NomeDaPropriedadeNaClasse, NomeDoCampoNoBanco, Tipo, Tamanho, Array(Opções)).

Veja que na linha 11 estamos mapeando a propriedade código para o campo contatoid no banco de dados. Informamos ainda que este é um campo inteiro de tamanho 11 e que como opção temos que ele é uma chave primária, não pode ser nulo e é autoincremento. As linhas 12, 13 e 14 fazem a mesma coisa, porém cada uma mapeando um campo diferente da tabela.

Feito isso o Lumine ainda pede que seja sobrescrito o método staticGet(). Este método é utilizado pelo Lumine para recuperar uma instância do objeto de forma estática utilizando o método get do próprio Lumine. Faz-se necessário sobrescrevê-lo para que possamos retornar a instância do objeto que estamos trabalhando no momento, no nosso caso, Contato. Isso é feito nas linhas 16 a 21. Por fim sobrescrevemos o destrutor da classe Lumine_Base (linhas 22 a 25). Repare que a única coisa que temos a fazer, fora as configurações, é informar os atributos da nossa classe. Isso pode ser visto entre as linhas 5 a 8.

Isso é o suficiente para que possamos criar um script onde instanciaremos um contato, passaremos os valores para ele e no final enviaremos para a base de dados. Só para conferir se está no caminho certo, a **Figura 3** exibe como deverá estar sua estrutura de pastas.

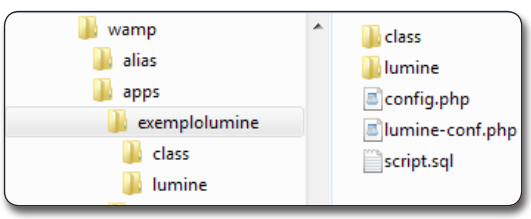


Figura 3. Estrutura final das pastas

Exemplo para testar o mapeamento

Agora só nos resta testar o nosso exemplo. Para isso vamos criar um arquivo index.php que terá a função de prover um formulário para cadastrar novos contatos no banco de dados. A **Listagem 5** mostra o conteúdo do script index.php.

Com exceção dos códigos HTML para gerar o formulário de cadastro, o restante do código é bastante simples. Primeiro iniciamos carregando o script config.php na linha 2. A seguir é realizada a verificação para saber se o formulário foi submetido (linha 3), nesse caso instanciamos um novo objeto Contato (linha 4), passamos os dados do formulário para o objeto (linhas 5 a 7) e invocamos o método save (linha 8). Isso é o suficiente para adicionar um novo contato no banco de dados. Na sequência inserimos o código HTML para gerar o formulário de cadastro (linhas 10 a 28).

Listagem 5. Script index.php

```
01 <?php
02 require_once dirname(__FILE__) . '/config.php';
03 if (isset($_POST['btnGravar'])){
04 $obj = new Contato();
05 $obj->nome = $_POST['nome'];
06 $obj->email = $_POST['email'];
07 $obj->telefone = $_POST['telefone'];
08 $obj->save();
09 }?>
10 <html>
11 <head>
12 <title>Cadastro de Contatos</title>
13 </head>
14 <body>
15 <form id="form1" name="form1" method="post" action="<?php echo $_SERVER
   ['PHP_SELF']; ?>">
16 <h1>Cadastro de Contatos</h1>
17 <table width="700" border="0" cellspacing="1" cellpadding="2">
18 <tr><td width="59%" valign="top">
19 <table width="100%" border="0" cellspacing="1" cellpadding="2"><tr>
20 <td width="36%">Nome</td>
21 <td width="64%"><input type="text" name="nome" id="nome" value="" />
   </td></tr>
22 <tr><td>E-mail</td>
23 <td><input type="text" name="email" id="email" value=""/></td></tr>
24 <tr><td>Telefone</td>
25 <td><input type="text" name="telefone" id="telefone"" /></td></tr>
26 <tr><td> </td>
27 <td><input type="submit" name="btnGravar" id="btnGravar" value=
   "Gravar" /></td></tr></table></td></tr>
28 <tr><td>
29 <?php
30 $lista = new Contato;
31 $lista->find();
32 Lumine_Util::showResult($lista);
33 ?>
34 </td></tr></table></form>
35 </body>
36 </html>
```

Um último detalhe fica por conta do método para listar os objetos já gravados no banco. Na linha 29 iniciamos novamente o script PHP, na linha 30 criamos um novo objeto contato e na linha 31 invocamos o método find(). Este método quando chamado desta forma recupera todos os objetos salvos no banco de dados. O segredo fica por conta do código da linha 32, onde fazemos uso da classe Lumine_Util. Através do showResult, exibimos a lista de objetos (passada como parâmetro) em forma de grid no browser.

Testando a aplicação

Para que possamos testar nossa aplicação precisamos criar um alias para o diretório onde salvamos nosso exemplo. Para nossa sorte o WAMPServer faz isso de uma forma muito simples. Para isso basta clicar no ícone do WAMPServer ao lado do relógio do Windows, selecionar o menu Apache, submenu Diretórios de Alias e por fim adicionar um alias. A **Figura 4** mostra o menu aberto.

Feito isso um prompt do DOS será aberto pedindo que você informe um alias, um apelido para o diretório onde está nossa aplicação. Digite exemplolumine, em minúsculo, e pressione enter. Agora o prompt irá pedir que você informe o caminho físico para onde este alias deverá apontar. Se você

manteve o diretório padrão mostrado na **Figura 2**, então informe C:/wamp/apps/exemplolumine e pressione enter. Pronto, alias configurado, podemos testar nossa aplicação. Abra o browser de sua preferência e acesse *http://localhost/exemplolumine*. A **Figura 5** mostra a aplicação em execução já com alguns dados inseridos.

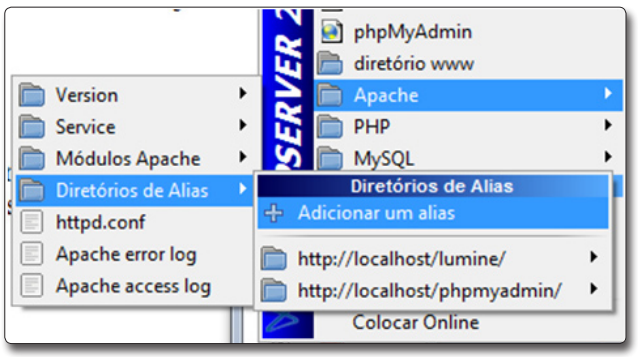


Figura 4. Criando um novo alias

Figura 5. Aplicação em execução

Conclusão

Ao utilizar um framework de persistência, passamos a usar boas práticas de desenvolvimento e programação orientada a objetos. Deixamos a SQL de lado, passamos a criar objetos ao invés de inserir registros. Passamos a nos concentrar mais nos problemas dos nossos clientes, que é o que realmente importa, do que nos problemas de infraestrutura da aplicação. Deixamos muitos dos detalhes inerentes à comunicação com o banco de dados por conta do framework. Passamos a aplicar em situações reais poderosos padrões de desenvolvimento. O Lumine, como vimos neste artigo, é um framework de mapeamento objeto / relacional poderoso para ser usado com o PHP, fácil de ser compreendido e utilizado, excelente para você desenvolver aplicações Web com elegância sem perder a produtividade. ●

Links

Download WAMPServer 2.0i
http://www.wampserver.com/en/download.php

Framework Lumine
http://www.hufersil.com.br/lumine

Documentação do Lumine
http://www.hufersil.com.br/Lumine-api-1.3.5.pdf

Blogs do autor
http://rodrigocarreiro.blogspot.com
http://twitter.com/carreiromourao

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!
Dê seu voto sobre este artigo, através do link:
www.devmedia.com.br/clubedelphi/feedback

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia group

Chegou o Sistema de Créditos DevMedia

Agora o **conteúdo completo** do nosso site está **ao seu alcance!**

A partir de agora todas as **4000 vídeo-aulas** do site DevMedia podem ser compradas individualmente.

Economia - Você não precisa mais assinar oito produtos diferentes para ter acesso ao conteúdo completo do site!

Todas as vídeos que você sempre quis ver a partir **R$ 0,99!**

Saiba mais sobre o Sistema de Créditos!

DevMedia GROUP

www.devmedia.com.br/creditos

Hospede seus sites de forma simples, rápida e ilimitada.

Plano **Básico**

- 1 domínio principal
- 50 domínios hospedados
- Transferência **ilimitada**
- Espaço **ilimitado**
- 10 contas de e-mail
- PHP, ASP e ASP.NET
- Painel de controle Plesk
- Suporte 24hs

Tudo isso por apenas

R$ **15,90** por mês
pagamento trimestral

Assine agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

- Conheça também nossos planos profissionais com registro de domínio grátis e isenção de taxas anuais enquanto for cliente.

HOSPEDIX
HOSPEDAGEM PROFISSIONAL
www.hospedix.com.br

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe 30 dias inteiramente grátis.

RVSTMEDIA

As promoções anunciadas não são cumulativas com outras promoções e são válidas apenas para clientes Hospedix ativos.